

CS SYSTEMES D'INFORMATION

TMA LID

Activité Défense, Espace et Sécurité
Division Espace
Département Services Spatiaux

PRS-MU-OASIS-0460-CS
Edition : 01 Date : 20/11/2013
Révision : 02 Date : 10/04/2014

Réf. : intentionnellement vide
Code diffusion : E

MANUEL D'UTILISATION

TUTORIEL API OASIS

Rédigé par : FERREIRA Jérôme REMISE Jean-Charles PASERO Sébastien	 DES/ESPACE/DSS DES/ESPACE/DSS DES/ESPACE/DSS	le : 01/06/2012	
Validé par : BATUT Françoise	 DES/ESPACE/DSS	le : 01/06/2012	
Pour application : CHOGNARD Catherine	 DES/ESPACE/DSS	le : 01/06/2012	

ZAC de la Grande Plaine, 5 Rue Brindejonc des Moulinais BP 15872, 31506 TOULOUSE Cedex 5

BORDEREAU D'INDEXATION

CONFIDENTIALITE : P

MOTS CLES : API OASIS, XIF, javadoc

TITRE DU DOCUMENT :

Manuel d'Utilisation
Tutoriel API OASIS

AUTEUR(S) :

FERREIRA Jérôme	DES/ESPACE/DSS
REMISE Jean-Charles	DES/ESPACE/DSS
PASERO Sébastien	DES/ESPACE/DSS

RESUME : Ce document est un tutoriel destiné à guider les développeurs JAVA pour l'utilisation de l'API OASIS.

DOCUMENTS RATTACHES : Ce document vit seul.

LOCALISATION :

VOLUME : 1	NBRE TOTAL DE PAGES : 97 DONT PAGES LIMINAIRES : 8 NBRE DE PAGES SUPPL. : 0	DOCUMENT COMPOSITE : N	LANGUE : FR

GESTION DE CONF. : NG

RESP. GEST. CONF. :

CAUSE D'EVOLUTION : Mise à jour de l'API suite à la DM BEST/FT/563 : XifChecker.

Complement d'informations concernant les namespace et prefixe d'un modèle suite à la DM BEST/FT/361.

Mise à jour de l'API suite à la DM BEST/FT/470 : XifExtensionUtilities.

Mise à jour de l'API suite à la DM BEST/FT/645 : XifConstant

CONTRAT : Néant

SYSTÈME HÔTE :

Microsoft Word 14.0 (14.0.7125)
C:\Program Files\GDOC\GDOC_4_2_2_2\GDOC\MODELES_GDOC\ModeleGDOCIndus.dot
Version GDOC : v4.2.2.2
Base projet : C:\Program Files\GDOC\TMA_LID.mdb

DIFFUSION INTERNE

Nom	Sigle	Bpi	Observations
BERRIRI Frédéric	DES/ESPACE/DSS		
FERREIRA Jérôme	DES/ESPACE/DSS		
ROUX Yann	DES/ESPACE/DSS		

DIFFUSION EXTERNE

Nom	Sigle	Observations
LARZUL Béatrice	DCT/PS/TVI	
POUPLARD Jean-Charles	DCT/PS/TVI	

MODIFICATION

Ed.	Rév.	Date	Référence, Auteur(s), Causes d'évolution
01	02	10/04/2014	intentionnellement vide FERREIRA Jérôme DES/ESPACE/DSS REMISE Jean-Charles DES/ESPACE/DSS PASERO Sébastien DES/ESPACE/DSS Mise à jour de l'API suite à la DM BEST/FT/563 : XifChecker. Complement d'informations concernant les namespace et prefixe d'un modèle suite à la DM BEST/FT/361. Mise à jour de l'API suite à la DM BEST/FT/470 : XifExtensionUtilities. Mise à jour de l'API suite à la DM BEST/FT/645 : XifConstant
01	01	20/11/2013	intentionnellement vide ROUX Yann DES/ESPACE/DSS FERREIRA Jérôme DES/ESPACE/DSS REMISE Jean-Charles DES/ESPACE/DSS Mise à jour de l'API suite à la DM BEST/FT/339
01	00	01/06/2012	intentionnellement vide ROUX Yann DES/ESPACE/DSS FERREIRA Jérôme DES/ESPACE/DSS Création du document

SOMMAIRE

FIGURES

GLOSSAIRE ET LISTE DES PARAMETRES AC & AD

API	Application Program Interface
APID	Application Process IDentifier
BEST	Beyond EAST
CCSDS	Consultative Committee for Space Data Systems
CNES	Centre National d'Etudes Spatiales
DEDSL	Data Entity Description Specification Language
DPE	Data Producer & Editor
EAST	Enhanced ADA SubseT
HTML	Hyper Text Markup Language
LID	Logiciel d'Ingénierie des Données
OASIS	Outil d'Aide à la Structuration d'Informations Spatiales
OCTAVE	Outils et Composants pour le Traitement, l'Acquisition et la Visualisation pour l'Exploitation de données
TC	Télécommande
XIF	XML Internal Format
XML	eXtensible Mark-up Language

Liste des paramètres AC :

Liste des paramètres AD :

1. GENERALITES

1.1. DOCUMENTS DE REFERENCE

DR1 Spécification du format XIF de l'outil OASIS Modeller
31/07/2012, Édit. 1, Rév. 5
EAST-IF-110-30038-CSSI

DR2 Spécification des extensions du format XIF
31/01/2013, Édit. 1, Rév. 11
EAST-IF-110-30042-CSSI

DR3 GUIDE D'UTILISATION DES EXTENSIONS DU XIF AU CNES
ROUX Yann, FAURE Ludovic, 18/02/2011, Édit. 01, Rév. 00
PRS-NT-BEST-0320-CS

DR4 Report concerning Space Data System Standards : The Data Description Language EAST - List of Conventions
01/05/1997, Édit. 1, Rév. 0
CCSDS646.0-G-1

1.2. DOCUMENTS APPLICABLES

2. INTRODUCTION

2.1. OBJECTIF

Ce document a pour objectif de présenter l'API OASIS et d'en définir les règles d'utilisation et de bon usage.

Cette API (*Application Programming Interface*) a été développée dans le langage JAVA afin de fournir un mécanisme de création/modification/lecture du format XIF. Elle a été développée, à l'origine, pour représenter l'interface de lecture/écriture du format XIF pour les outils de la suite BEST.

Ce document est structuré de la manière suivante :

- **Chapitre 3** : Ce chapitre présente un rappel succinct sur le format XIF et ses extensions en s'appuyant sur les documents de spécifications de ces formats,
- **Chapitre 4** : Ce chapitre présente la structure de l'API OASIS,
- **Chapitre 5** : Ce chapitre détaille les éléments structurants de l'API,
- **Chapitre 6** : Ce chapitre détaille la procédure d'import de l'API OASIS dans d'un projet Eclipse,
- **Chapitre 7** : Ce chapitre présente un ensemble de cas représentatifs d'utilisation de l'API.

2.2. APPLICATION

Ce document concerne **la version 3.1 du XIF.**

Ce document concerne **la version 2.4 des extensions du XIF**.

2.3. LIENS UTILES

2.3.1. Les contacts

Le support des utilisateurs des outils BEST ainsi que de l'API OASIS est assuré au travers de la mailing liste

east@cnes.fr

Cette liste comprend les personnes référents du CNES pour la suite d'outils BEST et de l'API OASIS ainsi que l'équipe en charge de la maintenance de ces outils.

2.3.2. Les sites Internet

Le site officiel de téléchargement des outils BEST, de l'API OASIS ainsi que de la documentation associée à ces outils est le suivant :

http://logiciels.cnes.fr

Le site officiel dédié aux produits LID du CNES est le suivant :

http://debat.c-s.fr

Le site suivant, en accès exclusivement depuis le réseau CNES offre une vision détaillée des outils proposés par le service et notamment les outils BEST :

http://prs.cnes.fr

3. LE FORMAT XIF

Le format XIF (XML Internal Format) est un standard CNES basé sur la syntaxe XML. Il a été défini à l'origine pour contenir l'ensemble des informations syntaxiques et sémantiques relatives aux recommandations EAST et DEDSL définies par le CCSDS.

Le format XIF a pour vocation première de décrire des flux de bits (données ASCII ou binaires) Il ne s'agit pas d'un format de données mais d'un modèle de description de données.

Le format XIF permet de décrire des structures de données sous forme d'arbres. Un fichier XIF peut contenir plusieurs arbres correspondant à plusieurs descriptions. Chaque arbre est rattaché à une racine et est composé d'éléments. Ces éléments sont définis à partir de types et de constantes. Ces types sont définis en utilisant les types de base suivants: entier, réel, caractère, énuméré, string, structure, tableau et liste. Les types et les constantes sont rattachés dans le XIF au même niveau que les racines des différents arbres.

La définition des éléments et des types du XIF peut être complétée grâce à une liste d'extensions. Une extension est un arbre XML susceptible de stocker n'importe quel type d'information. Par exemple dans le domaine du "Monitoring&Control", les alarmes définies au niveau d'un paramètre observable se présentent sous la forme d'une extension.

Le format XIF a pour vocation d'être le format d'échange de données d'un système spatial au sein du CNES. Il a déjà été adopté par plusieurs outils génériques CNES pour décrire ces données système : OASIS, OCTAVE, DPE, SOSIE, Atelier Données Système, etc.…

De par sa flexibilité et son adaptabilité, le format XIF offre une compatibilité avec différents standards ou format généralement en relation avec le domaine spatial. La figure suivante donne un aperçu des transformations possibles (à l'aide de passerelles de conversions) :

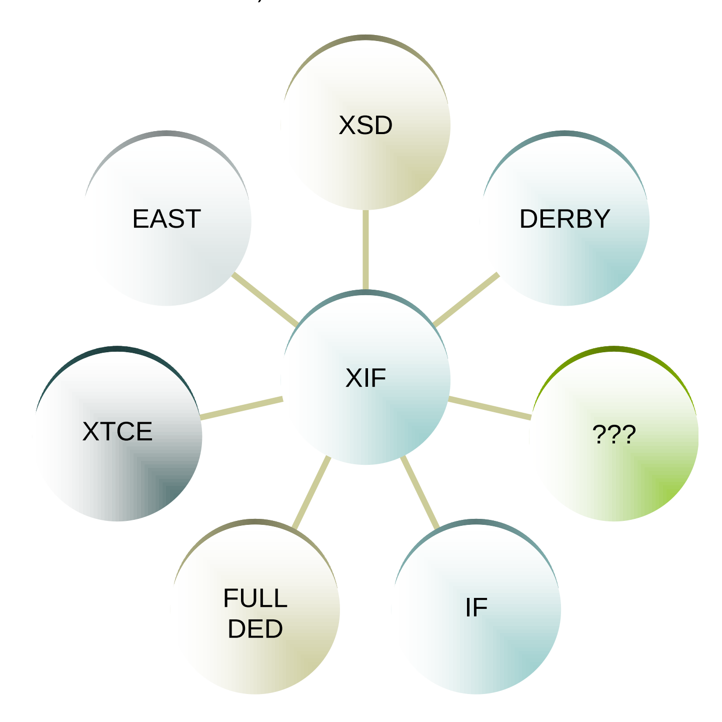


Figure 1 XIF format flexible

Pour rappel, le format XIF est spécifié dans le document suivant :

- DR1 : Format XIF d'OASIS Modeller EAST-IF-110-30038- CSSI. Ce document est également référencé au RNC (RNC-CNES-E-TM-80-501).

Sa grammaire est également décrite de façon formelle grâce à son schéma XML qui est également disponible (CF. DR1).

Les extensions sont décrites dans le document suivant :

- DR2 : Spécification extensions du format XIF EAST-IF-110-30042-CSSI.

Sa grammaire est également décrite de façon formelle grâce à son schéma XML qui est également disponible (CF. DR2).

4. STRUCTURE DE L'API OASIS

4.1. PRESENTATION DE L'API OASIS

L'API OASIS est une librairie développée dans le langage JAVA et offrant une interface de lecture, d'écriture et de validation du format XIF et de ses extensions.

Note : l'API est optimisée pour la version 1.6 du JDK.

Cette API OASIS se présente sous la forme d'un ensemble de répertoires dont la structure est représentée ci-dessous :

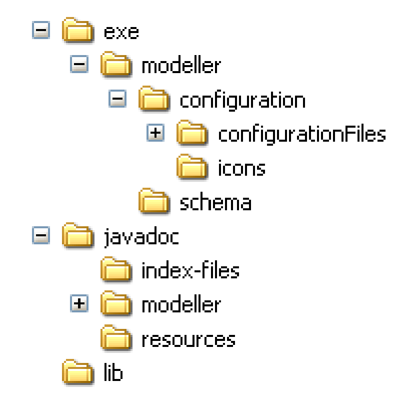


Figure 2 Organisation de l'API OASIS

Le répertoire « **exe** » contient le fichier JAR du « modeller » nommé « modeller.jar » ainsi que les fichiers de configuration du « modeller ». Il s'agit de l'API du XIF.

Ce fichier « modeller.jar » contient l'essentiel des classes JAVA permettant la manipulation des fichiers XIF.

Le répertoire « schema » contient les schémas XML sur lesquels s'appuie le JAR du « modeller » pour définir sa propre configuration. Ce répertoire contient également le schéma XML du format XIF nommé « xif.xsd ».

Le répertoire « configuration » contient l'ensemble des fichiers de configuration du « modeller ». Il contient notamment le fichier « userSettings.xml » permettant de sauvegarder et d'initialiser la configuration de l'IHM de l'outil OASIS Modeller contenu dans l'API. Il contient également le fichier « xifExtensions.xsd » qui définit l'ensemble des extensions du format XIF prédéfinis et qui peut être enrichi par les utilisateurs.

Le répertoire « **lib** » contient l'ensemble des librairies annexes (.jar) nécessaires au fonctionnement de l'API OASIS. Ces librairies (.jar) sont emportées par l'API car elles sont nécessaires au fonctionnement de celle-ci et doivent être référencées dans le « classpath » des applications faisant appel à l'API OASIS.

Un répertoire « **javadoc** » contenant l'ensemble de la documentation JAVA de l'API au format HTML et générée à partir des commentaires du code du « modeller ». Cette documentation JAVA est très utile dans le développement d'application JAVA s'appuyant sur l'API OASIS car elle détaille l'ensemble des paquetages et classes constituant l'API.

4.2. ARCHITECTURE DE L'API OASIS

4.2.1. Librairie JAXB

L'API d'OASIS repose sur la librairie JAXB. Celle-ci permet de générer, à partir du schéma définissant la grammaire du XIF, les classes nécessaires à la construction du modèle de données XIF.

JAXB se présente sous la forme de librairies JAVA permettant de manipuler les fichiers XML et fournissant un support des technologies telles que SAX, DOM, XSLT, ... La documentation officielle est disponible sur le site de Sun à l'adresse suivante : http://java.sun.com/xml/jaxb/

JAXB permet de produire automatiquement des classes de mapping capable de lire et charger des documents XML en mémoire, et aussi, capable de stocker la structure d'objets dans un document XML qui reflète l'état des objets en mémoire.

La figure suivante permet d'illustrer le mécanisme fourni par JAXB :

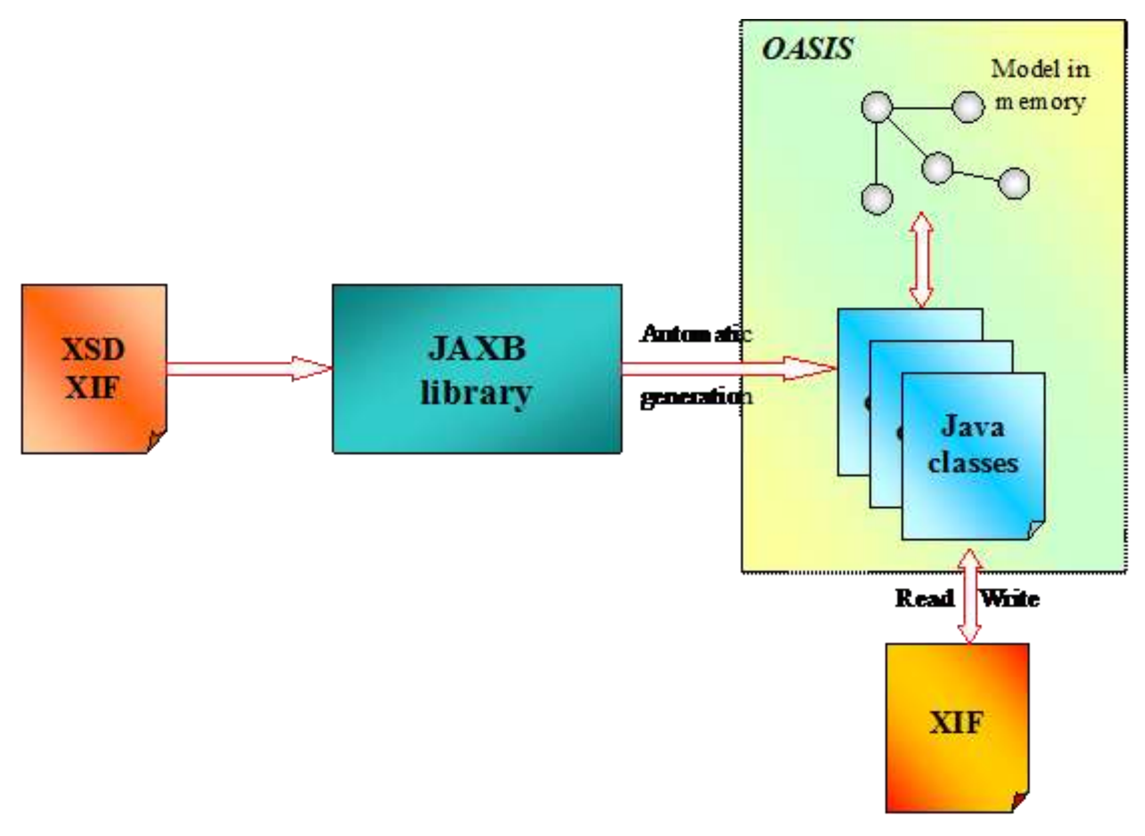


Figure 3 Mapping XML <=> JAVA

4.2.2. Modèle interne de l'API OASIS

Le cœur de l'API OASIS s'articule autour du schéma XML du format XIF. Le modèle de données qui permet de lire et d'écrire les fichiers XIF est généré à partir de l'outil XJC sur la base de l'API JAXB (Cf. 4.2.1).

Le schéma ci-dessous illustre l'architecture du cœur de l'API qui se trouve dans le paquetage :

```
package modeller.core.dataManager.xifManager.api;
```

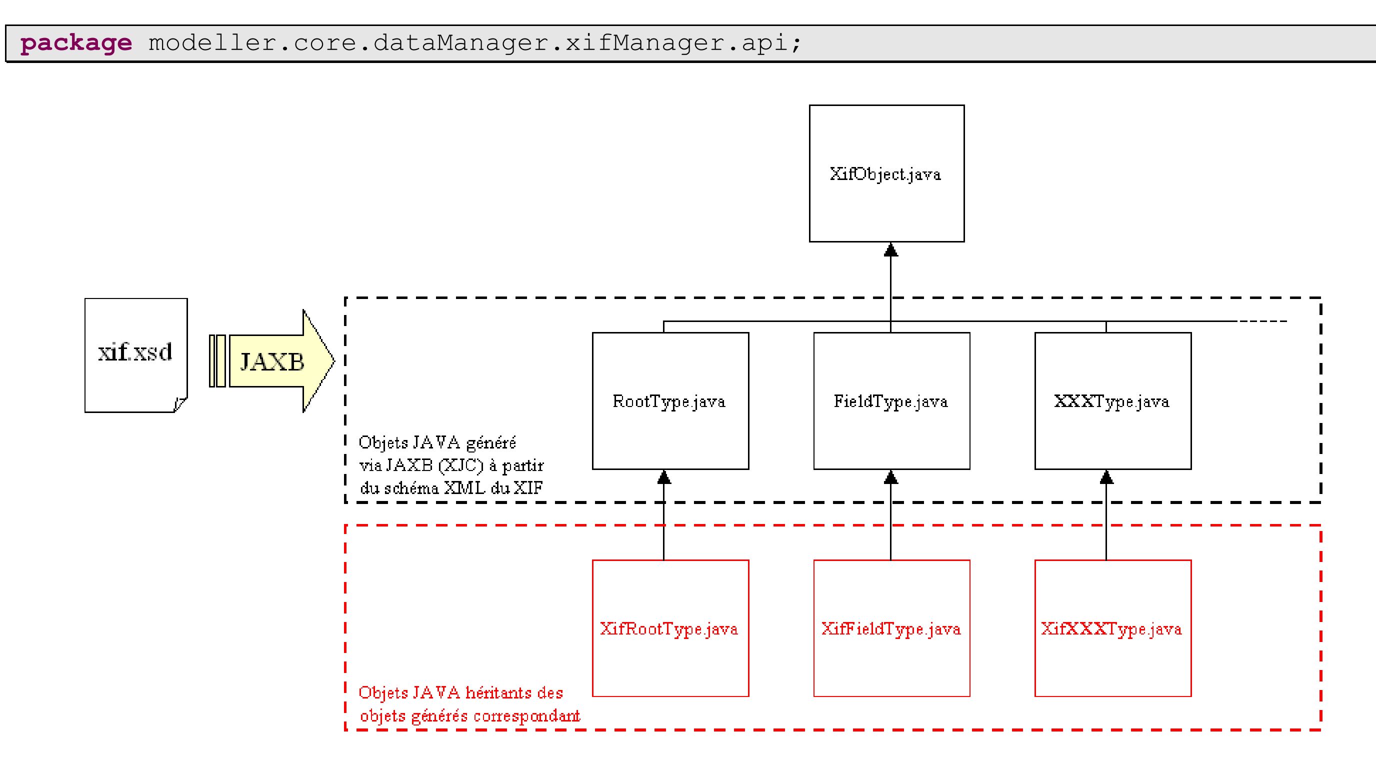


Figure 4 Modèle de donnée du XIF

L'ensemble des classes JAVA générées via JAXB hérite de la classe « XifObject ». Cette classe offre un ensemble de méthodes communes à toute classe composant le modèle XIF.

Dans le schéma ci-dessus, la chaîne « XXX » représente un type du format XIF (Integer, Real, Character, String, Record, …). Chacun de ces types est donc modélisé en mémoire par un objet JAVA qui en contiendra les informations lors du chargement du fichier XIF. Ces classes sont toutes dérivées dans l'API OASIS (en rouge dans le schéma ci-dessus) par les classes nommées de la même façon avec le préfixe « Xif ». Ces classes filles offrent un ensemble de méthode « métier » facilitant grandement l'utilisation du modèle XIF en mémoire.

Les classes générées via l'API JAXB sont stockées dans le paquetage « modeller.core.dataManager.xifManager.api.jaxbObjects ». Les classes qui en héritent (en rouge dans le schéma) sont stockées dans le paquetage « modeller.core.dataManager.xifManager.api ».

Note : Lors de la manipulation du modèle de donnée XIF en mémoire, seules les classes se trouvant dans le paquetage « modeller.core.dataManager.xifManager.api » doivent être utilisées. Il est fortement recommandé de ne pas manipuler directement les classes produites par JAXB.

Note : l'API fournit cette couche d'interface entre le modèle de données JAVA et le format XIF mais elle ne fournit pas d'outil de manipulation des extensions. En effet, une extension étant un arbre XML susceptible

de stocker n'importe quel type d'information, l'API OASIS ne peut pas connaître ni fournir d'objets spécifiques de manipulation des données contenues dans les extensions. Comme nous le verrons dans les exemples présentés dans les chapitres suivants, il est conseillé d'utiliser l'API DOM pour parcourir, créer, modifier ou supprimer les éléments des extensions.

4.2.3. Architecture globale de l'API OASIS

L'architecture globale de l'API d'OASIS se présente comme suit :

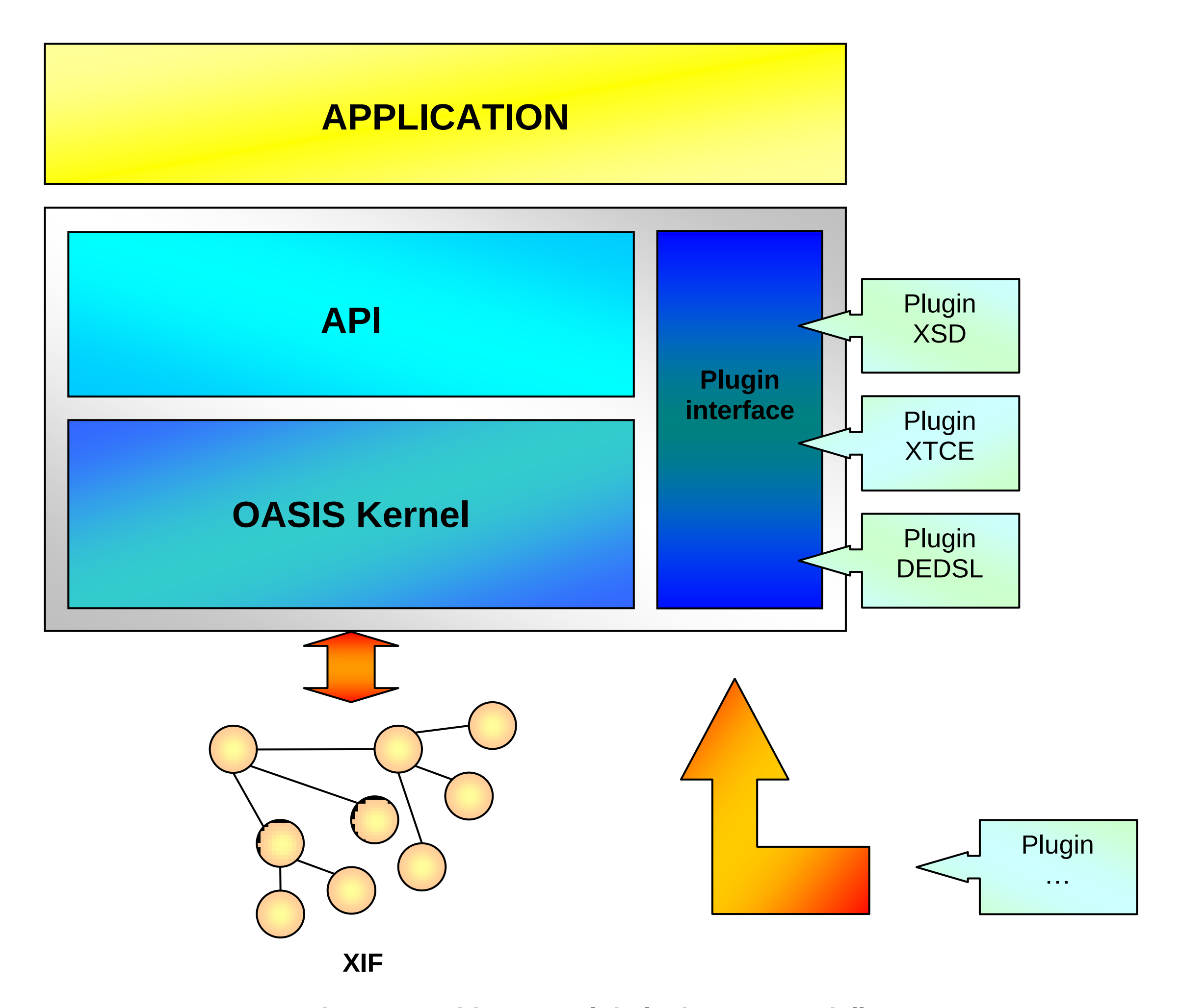


Figure 5 Architecture globale d'OASIS Modeller

Comme le montre le schéma précédent, il s'agit d'une architecture modulaire basée sur le mécanisme des plug-ins. Les différentes parties sont décrites ci-dessous :

- XIF : il s'agit du format de sauvegarde du modèle interne de l'outil OASIS Modeller afin d'assurer la persistance. Celui-ci offre une description commune des données qui peut être élargie à volonté grâce à un mécanisme d'extension.

- OASIS kernel : Cette couche est responsable de la manipulation et de la gestion du modèle interne. Le noyau a pour rôle de lire et écrire le fichier « XIF », de le modifier, de gérer les librairies de types…

- API (Application Programming Interface) : interface programmatique qui permet la communication entre le noyau et l'IHM ainsi que la liaison entre le noyau et la plug-in interface.
- Plugin interface : interface commune à tous les plug-ins qui a pour rôle de fournir les services nécessaires à l'interaction entre les différents plug-ins et l'outil OASIS Modeller. De plus, de nouveaux plug-ins implémentant l'interface ainsi définie pourront être ajoutés dans l'outil OASIS Modeller,
- Plugin : il s'agit de « briques » indépendantes les unes des autres et qui ont pour rôle une fonction bien précise. Les plug-ins permettent d'étendre les fonctionnalités du noyau d'OASIS (nouvelles entrées/sorties, personnalisation des IHM…).

4.2.3.1. Décomposition

L'API OASIS est composé de 4 modules principaux comme l'illustre la figure suivante :

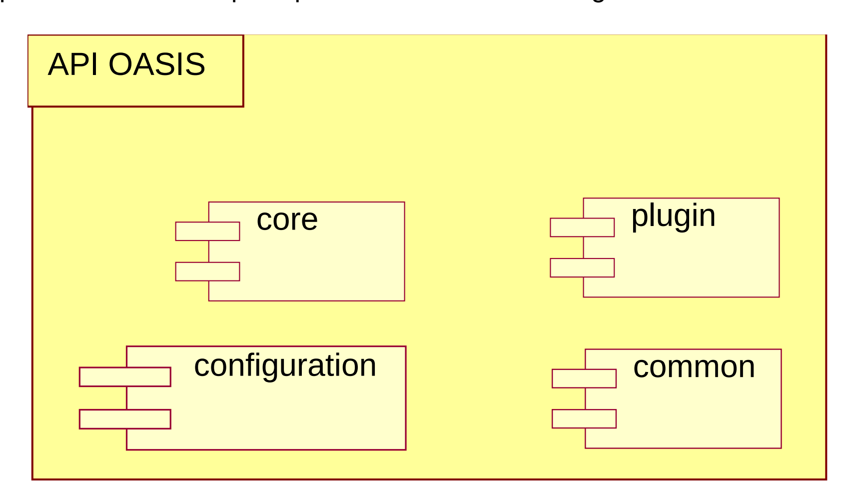


Figure 6 Vue composant globale

Chacun des composants est implémenté dans le paquetage **modeller.***. La figure suivante décrit les différents paquetages :

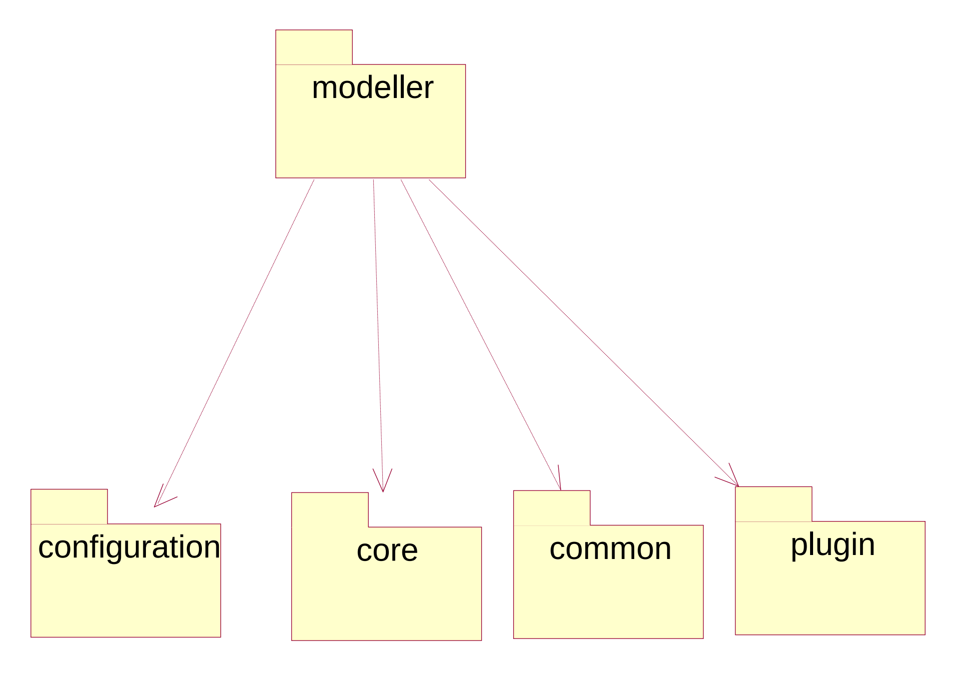


Figure 7 Vue paquetage globale

Le tableau qui suit présente la correspondance entre les composants logiques et leurs implémentations sous formes de paquetages :

Composant	Rôle	Paquetage associé
plugin	Permet de gérer, manipuler, installer les différents plug-ins qui viennent étendre les fonctionnalités de l'outil OASIS Modeller.	modeller.plugin.*
common	Fournit un ensemble de classes utilitaires communes à tous les paquetages de l'outil OASIS Modeller.	modeller.common.*
core	Permet de gérer et manipuler le format interne : fichiers XIF (.xif). Il fournit également des fonctionnalités de recherche.	modeller.core.*
configuration	Contient l'ensemble des classes permettant la configuration de l'outil OASIS Modeller : fichiers de ressources, langue...	modeller.configuration.*

4.2.4. Focus : le paquetage « core »

4.2.4.1. Fonctions

Le noyau de l'API OASIS, représenté par le paquetage « core » décrit précédemment, a pour rôle principal de gérer le modèle interne. Il a en charge :

- Lecture et sauvegarde du fichier « xif » définissant le modèle,
- Création de l'arbre de description en mémoire (à partir d'un fichier « xif » lu),
- Manipulation de l'arbre de description en mémoire : ajout, suppression, modification, des champs ou/et de leur type,
- Gestion des sous-XIF,
- Contrôle de cohérence du modèle en mémoire,
- Gestion des tailles dynamiques pour les tableaux et chaînes de caractères,
- Gestion des conditions d'existence,
- Gestion de la configuration sémantique basée sur le DEDSL,
- Gestion du module physique contenant les différentes machines avec l'implémentation des réels ou entiers codé machine.

4.2.4.2. Architecture

La figure suivante présente les différents composants implémentés dans le paquetage **modeller.***.

La figure suivante décrit les différents paquetages :

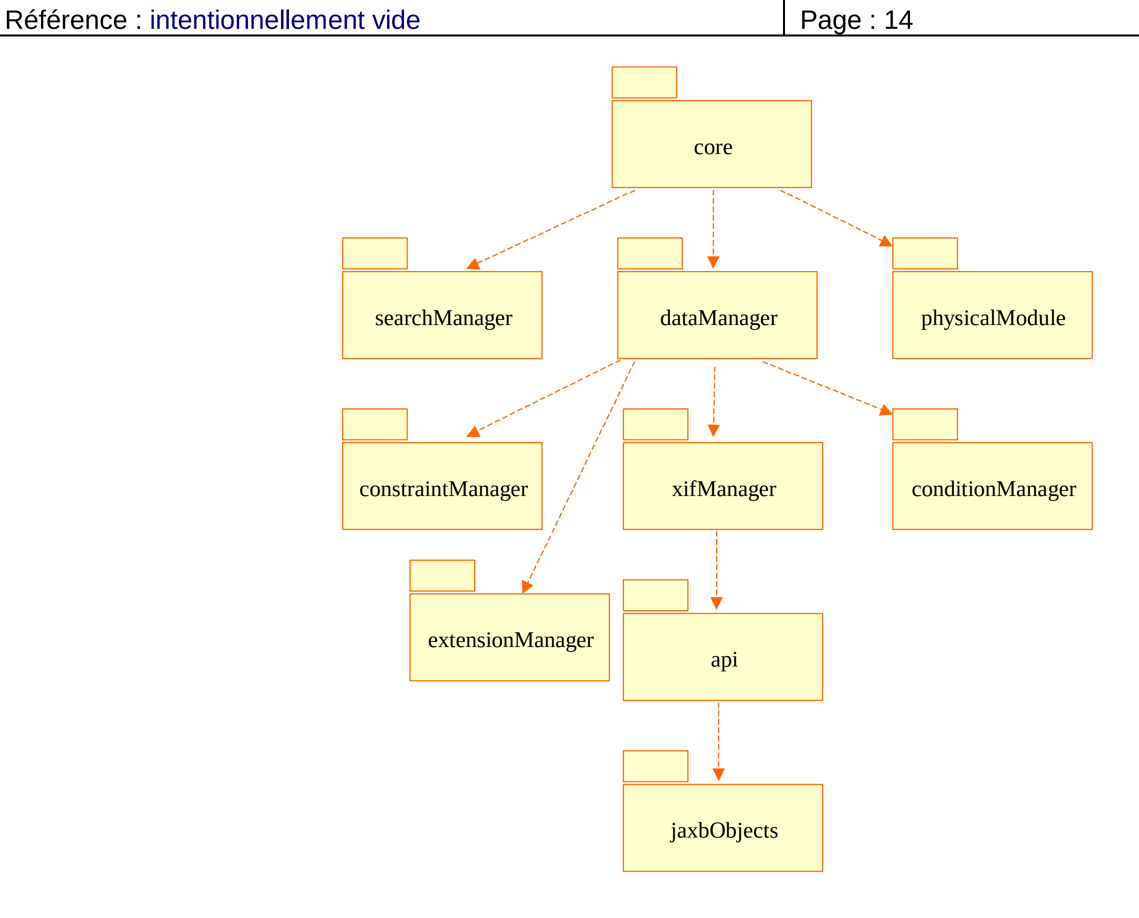


Figure 8 Décomposition en paquetage du noyau

La table suivante décrit les paquetages ainsi que les classes qu'ils contiennent :

Paquetage	Description	Classes
modeller.core.dataManager.*	Contient l'ensemble des classes qui permettent de manipuler le modèle interne, les librairies de types.	DataManager.
modeller.core.searchManager.*	Ensemble de classes permettant d'effectuer une recherche sur le modèle interne.	DataSearch…
modeller.core.physicalModule.*	Ce paquetage a en charge de gérer l'ensemble des conventions d'encodage associé aux machines	Machines, Norms, BoundValue, Binreps, ElementOfInteger, Configuration, ElementBinrep, ElementMachine,
modeller.core.dataManager.extensionManager.*	Contient la classe permettant de manipuler les extensions. Plus précisément cette classe représente le modèle XSD des extensions en mémoire.	ExtensionModel.
modeller.core.dataManager.constraintManager.*	Classes permettant de définir des contraintes sur le XIF en spécifiant les discriminants.	ConstraintManager, ConstraintFileHandler, Constraint…
modeller.core.dataManager.xifManager.*	Classes fournissant les services de manipulation et de gestion du modèle interne de données.	XifDataModel, XifDataModelHandler, XifChecker, XifDiscriminantAssociation, …

Paquetage	Description	Classes
modeller.core.conditionManager.*	Contient l'ensemble de classes et paquetages permettant de manipuler les conditions de taille et d'existences.	ConditionManager…
modeller.core.dataManager.xifManager.api.*	Ensembles de classes qui composent les éléments et types du modèle interne et représentant l'API du modèle interne.	XifArrayType, XifListType, XifRecordType, XifIntegerType, XifEnumeratedType, XifGenericType, XifStringType, XifConstant, XifCharacterType, XifField, XifRoot.…
modeller.core.dataManager.xifManager.api.jaxbObjects.*	Ensemble de classes générées à l'aide de JAXB à partir du schéma de définition du modèle et permettant la lecture et l'écriture du xif.	Ces classes sont automatiquement générées par JAXB.

4.2.5. La cohérence des données

L'API OASIS a été créé pour fournir une couche de manipulation du modèle de données XIF et ainsi en assurer un niveau de cohérence. A ce titre, elle assure la cohérence dans le référencement des éléments. Par exemple, lorsque l'on modifie le nom d'un type interne via les méthodes dédiées de l'API OASIS, celle-ci assure que les éléments de l'arbre XIF qui référencent ce type sont modifiés pour qu'ils référencent toujours le type renommé.

C'est l'ensemble des classes qui composent le paquetage « modeller.core.dataManager.xifManager » qui assure la cohérence de ces données et dont l'utilisation est détaillée dans ce document.

En revanche, cette cohérence n'est pas assurée pour les extensions. En effet, les extensions définissent un ensemble d'informations méconnues de l'API OASIS, la cohérence de ces informations ne peut donc pas être assurée. C'est donc au système qui utilise l'API OASIS d'assurer la cohérence des données de ces extensions.

Si l'on prend donc l'exemple d'une fonction de transfert, dans le cas où la fonction de transfert est renommée dans la liste des « calibrator » ou « decalibrator », c'est au système embarquant l'API OASIS de modifier la référence sur les paramètres du modèle de données qui référencent cette fonction.

5. DETAIL DES ELEMENTS STRUCTURANTS DE L'API

Ce chapitre présente le détail des paquetages et des classes qui composent l'API OASIS. Chaque classe et chaque méthode de l'API ne pouvant être détaillée dans ce chapitre, une présentation générale est faite pour les paquetages de l'API puis une présentation plus détaillée est effectuée pour les classes les plus importantes et les plus pertinentes de l'API.

Il est cependant fortement conseillé aux utilisateurs de cette API de consulter la documentation JAVA associée à l'API OASIS (« javadoc ») qui détaille chaque paquetage, classe et méthode et fournit un support essentiel aux développements utilisant cette API.

5.1. LE PAQUETAGE MODELLER.*

Ce paquetage est le paquetage de base du JAR du « modeller ». Il contient le point d'entrée de l'API OASIS, c'est-à-dire la classe d'interface nommée « ModellerServicesInterface ».

Les paquetages contenus dans le paquetage « modeller » sont décrits dans les paragraphes suivants.

5.1.1. La classe d'interface « ModellerServicesInterface »

Cette classe contient 4 méthodes qui constituent les appels principaux à réaliser pour l'écriture ou la lecture d'un fichier XIF :

```
public static XifResource loadXmlIf(String pathname) throws Exception
```

Cette méthode permet de lire et de charger en mémoire un fichier XIF.

Elle prend en paramètre une chaîne de caractère constituant le chemin absolu du fichier XIF que l'on souhaite charger en mémoire et retourne un objet « XifResource » qui constitue l'objet de manipulation du descriptif XIF chargé en mémoire.

```
public static XifResource loadXmlIf(String pathname, HashMap<XifReference, String>
unloadableRefs) throws Exception
```

Cette surcharge de la methode précédente a un rôle identique mais permet, en plus, de recueillir toutes les références entre modèles de la ressource qui n'ont pas pu être établies, avec le message d'erreur correspondant associé, placées dans la Map en second paramètre. Pour une description plus précise des références, se référer au chapitre 7.1.2.

```
public static boolean generateFile(String aPluginId, String aFileType, XifResource
aResource, String anOutputFile, XifNode aRoot)
```

Cette méthode permet de générer un fichier d'un certain type à partir d'un ensemble de paramètres définis.

Cette méthode ne peut être utilisée qu'en association avec le plug-in OASIS correspondant au premier paramètre passé à la méthode. Il est donc nécessaire que le plug-in correspondant soit placé dans le « répertoire modeller/plugins » de l'API pour que cette méthode fonctionne.

```
public static boolean importFile(String aPluginId, String aFileType, String anInputFile, String anOutputFile)
```

Cette méthode permet de générer un fichier XIF à partir d'un ensemble de paramètres définis.

Cette méthode ne peut être utilisée qu'en association avec le plug-in OASIS correspondant au premier paramètre passé à la méthode. Il est donc nécessaire que le plug-in correspondant soit placé dans le « répertoire modeller/plugins » de l'API pour que cette méthode fonctionne.

5.2. LE PAQUETAGE MODELLER.COMMON.*

Le paquetage « modeller.common » est un paquetage contenant un ensemble de classes utilitaires offrant des services de manipulation de données, de recherches, de validation …

Ce paquetage est très utile dans la manipulation des données XIF et offre une classe de manipulations des extensions via l'API DOM.

Classe	Rôle
CharacterCheck	Fournit un ensemble de méthodes statiques permettant de valider la conformité d'un caractère en fonction d'un contexte donné, si un caractère est imprimable, etc.
Check	Fournit un ensemble de méthodes statiques permettant de vérifier le contenu d'une chaîne de caractère et de valider son formalisme en fonction d'un contexte donné.
Utilities	Fournit un grand nombre de méthodes statiques utiles pour la manipulation des éléments du modèle de données XIF.
XifExtensionUtilities	Fournit un grand nombre de méthodes statiques utiles pour la manipulation et la validation des extensions des éléments du modèle de données XIF.

5.3. LE PAQUETAGE MODELLER.CONFIGURATION.*

Ce paquetage contient un certain nombre de fichiers de configuration issus de la recommandation CCSDS « EAST conventions » (description physique des types entiers et réels selon la machine (SUN, PC, …) induisant une écriture différente des bits et octets soit LITTLE-ENDIAN soit BIG-ENDIAN, et la norme d'encodage (IEEE, …) induisant une écriture différente des mantisses et exposants des réels sur le même nombre d'octets.

5.3.1. La classe « ModellerConfiguration »

Cette classe permet de charger la configuration de l'API OASIS et d'initialiser un grand nombre de paramètres de configuration utiles pour l'API.

Il est donc essentiel que la méthode « loadModellerConfiguration() » soit appelée à l'initialisation de tout

programme utilisant l'API OASIS.

Dans le cas de la lecture d'un fichier XIF via l'interface « ModellerServiceInterface », cette méthode est appelée dans le code de la méthode « ModellerServicesInterface.loadXmlIf », il n'est donc pas nécessaire d'y faire appel.

En revanche, à la création d'un fichier XIF, comme on le verra dans les chapitres suivants, il est essentiel d'y faire appel.

5.4. LE PAQUETAGE MODELLER.CORE.DATAMANAGER.*

Le paquetage « dataManager » est le composant principal du « core » puisque il a en charge la gestion et la manipulation du modèle interne de données.

Ce composant contient la classe « DataManager » qui offre l'ensemble des services permettant de gérer le modèle interne « xif » de l'outil OASIS Modeller. Les sous paquetages sont décrits dans les paragraphes qui suivent.

5.4.1. Le paquetage « modeller.core.dataManager.xifManager »

Le modèle interne de l'outil OASIS Modeller est basé sur le format XML. Ce format « xif » est géré par le composant « xifManager » qui a pour rôle de transformer le fichier « xif » en une arborescence d'objets et inversement lors de la sauvegarde de la description de données.

Ce paquetage offre aussi un ensemble de services permettant de manipuler le modèle de données en mémoire :

- Ajout/suppression d'éléments dans l'arbre,
- Insertion d'éléments,
- Modification des informations sémantiques et syntaxiques.
- Vérification du modèle

5.4.1.1. Le paquetage « modeller.core.dataManager.xifManager.api »

Le paquetage « api » fournit l'ensemble des services permettant à l'utilisateur d'éditer/modifier le modèle interne de données.

Ce composant est composé des classes suivantes :

- XifArrayType et XifListType : classe permettant de manipuler et gérer les répétitions d'éléments sous forme de tableaux ou de liste,
- XifRecordType : type de structure permettant de hiérarchiser la description sous forme arborescente,
- XifIntegerType, XifStringType, XifRealType, XifCharacterType, XifEnumeratedType : ensemble de classes représentant respectivement les types simples entier, chaîne de caractère, décimal, caractère et énuméré,
- XifGenericType : cette classe permet de manipuler des types simples non pris en compte par les types listés précédemment,

- XifConstant : classe permettant de manipuler les constantes utilisées dans la description.

5.4.1.2. Le paquetage « modeller.core.dataManager.xifManager.api.xifObjects »

Ce paquetage contient l'ensemble des classes nécessaires à l'écriture et à la lecture du fichier « xif » représentant la description de données.

Ces classes sont générées de façon automatique grâce à la technologie Java appelée JAXB (Java API for XML Binding).

A partir du schéma définissant le format « xif », la librairie JAXB génère les classes Java permettant de lire et écrire le fichier au format « xif ».

5.4.2. Le paquetage « modeller.core.dataManager.conditionManager »

Le paquetage « conditionManager » est responsable de la gestion des discriminants. Ces derniers peuvent représenter soit des tailles dynamiques de tableaux par exemple, soit des conditions d'existence d'éléments.

Les services suivants sont fournis :

- Gestion des tailles dynamiques qui peuvent être d'ordre simple ou calculées,
- Gestion des conditions d'existence soit simples ou calculées,
- Gestion d'un arbre synthétique des conditions permettant de condenser et d'ordonner les conditions d'existence,
- Évaluation des expressions afin de déterminer la taille (cas des tableaux) ou la présence (cas des conditions d'existence) des éléments de l'arbre.

5.4.3. Le paquetage « modeller.core.dataManager.extensionManager »

Le paquetage « extensionManager » fournit une classe de la gestion générique d'un schéma XSD d'extensions. Elle peut être utilisée dans le parcours de l'arbre XSD.

5.4.4. Le paquetage « modeller.core.dataManager.constraintManager »

Le paquetage « constraintManager » est responsable de la gestion des contraintes associées à un modèle de données pour en définir le découpage en s'appuyant par exemple sur les conditions d'existences.

5.5. LE PAQUETAGE MODELLER.CORE.PHYSICALMODULE.*

Le paquetage « physicalModule » a pour rôle de gérer les conventions utilisées pour encoder les entiers et réels lors de la génération de fichier de sortie par l'outil OASIS Modeller.

Ce composant fournit donc un ensemble de services permettant à partir d'une machine donnée de déterminer les différentes normes associées pour l'écriture des entiers ou des réels.

L'ensemble des conventions ainsi définies respecte le document CCSDS DR4.

La figure suivante décrit le diagramme de classe associé à ce paquetage :

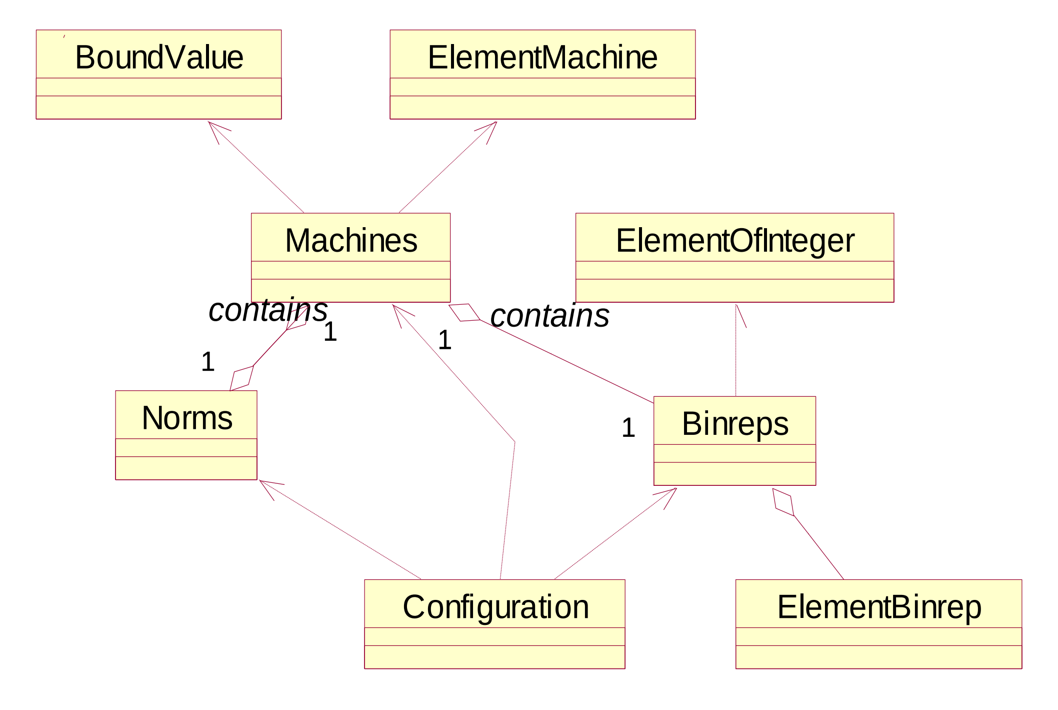


Figure 9 diagramme de classes du « physicalModule »

5.6. LE PAQUETAGE MODELLER.CORE.SEARCHMANAGER.*

Le paquetage « searchManager » offre une API permettant de réaliser des recherches dans le modèle interne de données.

Cette recherche peut porter soit sur les informations syntaxiques, soit sur les informations sémantiques.

- La recherche syntaxique est basée sur les informations suivantes : nom de l'élément, type, taille ou bornes,
- La recherche sémantique est basée sur le nom des attributs sémantiques ou leur contenu.

5.7. LE PAQUETAGE MODELLER.PLUGIN.*

Le système de plugin permet principalement l'ajout ou le remplacement de nouvelles fonctions dans l'outil OASIS Modeller. Ce mécanisme permet aux utilisateurs d'ajouter, étendre, mettre à jour l'outil OASIS Modeller de façon transparente. Cette propriété permet ainsi d'étendre indéfiniment les possibilités de l'outil OASIS Modeller.

Les classes principales qui composent le système de plugins sont :

- La classe « ModellerPluginLoader » : celle-ci est responsable du chargement en mémoire des plugins. Ces derniers sont chargés dynamiquement dans la même JVM que l'outil OASIS Modeller,
- La classe « ModellerPluginManager » : cette classe gère l'ensemble des plugins chargés dans l'outil OASIS Modeller,
- le « ModellerPluginInterface » : il s'agit en fait d'une interface qui permet la communication et le contrôle entre l'outil OASIS Modeller et le plugin. Tous les plugins doivent implémenter cette interface.

Les plugins, pour pouvoir être reconnus et chargé en mémoire de manière transparente par l'outil OASIS Modeller, doivent respecter une certaine implémentation.

6. IMPORT DE L’API OASIS DANS UN PROJET ECLIPSE

La procédure nécessaire à l’import de l’API OASIS dans un projet Eclipse est la suivante :

1. Créer un projet sous Eclipse ou ouvrir le projet dans lequel on souhaite utiliser l'API.

2. Décompresser l'archive API_OASIS_x.y.zip dans le répertoire du projet Eclipse où X.Y représente la version de l’API concernée.

3. Dans les propriétés du projet, dans le "java build path", ajouter l'ensemble des librairies (.jar) contenues dans le répertoire « lib » et ses sous répertoires ainsi que le JAR "modeller.jar" contenu dans le répertoire « exe ».

4. Il est nécessaire d'ajouter une variable d'environnement à l'appel de la machine virtuelle ou de la commande « JAVA » permettant de définir le chemin de l'environnement du « modeller » pour sa propre configuration : **-Dbest.home=<path_projet_eclipse>\exe**

7. EXEMPLES REPRESENTATIFS D'UTILISATION

7.1. LECTURE D'UN FICHIER XIF

L'ensemble du code sur lequel s'appuie l'exemple présenté dans ce chapitre se trouve en Annexe A de ce document.

7.1.1. Lecture d'un fichier XIF et récupération des données

7.1.1.1. Chargement du fichier XIF dans le modèle interne de l'API

Le point d'entrée principal de la lecture d'un fichier XIF est assuré par la classe « ModellerServicesInterface » qui offre une méthode statique de lecture d'un fichier XIF.

Cette méthode prend en paramètre une chaîne de caractère constituant le chemin absolu du fichier XIF que l'on souhaite charger en mémoire et retourne un objet « XifResource » qui constitue l'objet de manipulation du descriptif XIF chargé en mémoire.

Exemple d'utilisation :

```
String fichierXif = "d:\\un_projet_best\\un_fichier.xif";
XifResource xifResource = ModellerServicesInterface.loadXmlIf(fichierXif);
```

7.1.1.2. Vérification du modèle

Le paquetage « modeller.core.dataManager.xifManager » contient la classe « XifChecker » qui fournit la méthode « validate(boolean checkAllExtensions) » permettant de vérifier certains critères tels que :

- Les conditions d'existence
- Les extensions des nœuds de l'arbre
- Les types référencés
- Les tailles de tableaux et de chaînes de caractères
- Les bornes des types entiers
- Les fils des types RECORD

Ce comportement par défaut peut être complété, grâce à la valeur du paramètre d'entrée, d'une vérification syntaxique de toutes les extensions du modèle, incluant :

- Les informations générales
- Les champs globaux
- Les types internes
- Les constantes

Il suffit d'appeler la méthode « validate(boolean checkAllExtensions) » avec la valeur « true ».
Appeler la méthode « validate() » de cette même classe équivaut à appeler la méthode « validate(boolean checkAllExtensions) » avec la valeur « false ».

Exemple d'utilisation

```
XifChecker checker = xifResource.getXifDataModel().getXifChecker();
```

```
checker.validate(true); // vérification syntaxique de toutes les extensions

if (!checker.isValid()) {
   for(XifMessage errorMessage : checker.getErrorMessages()){
      System.out.println(errorMessage.toString());
   }
}
```

7.1.1.3. Parcours des arbres du modèle

Le parcours des arbres ou l'accès à un des éléments « racine » XIF s'effectue via le modèle principal du XIF, c'est-à-dire la classe « XifDataModel » correspondant au modèle interne du XIF « maître » chargé en mémoire. Cette instance est récupérée via la resource « XifResource » qui porte l'ensemble du modèle de données chargé en mémoire correspondant au fichier Xif chargé et ses références. (Voir le chapitre 7.2.1.2 pour plus d'informations)

```
XifNode root = null;
int nbRoots = xifResource.getXifDataModel().getRootCount();
for (int i = 0 ; i < nbRoots ; i++) {
      root = xifResource.getXifDataModel().getRootAt(i);
      System.out.println("processing root named = " + root.getName());
      // Manipulation du nœud root...
}
```

Une fois le nœud « racine » récupéré, il est possible de parcourir les éléments de l'arbre pour récupérer les informations du chaque nœuds ou chaque feuille de l'arbre. Pour cela, l'API a été conçue de façon à parcourir les nœuds comme des éléments d'un arbre DOM, ces nœuds « XifNode » embarque les informations de chaque éléments qui ont été construit au chargement du fichier XIF en mémoire.

Le schéma ci-dessous illustre l'architecture des classes JAVA de l'API montées en mémoire et embarquant les informations de chaque nœuds de l'arbre :

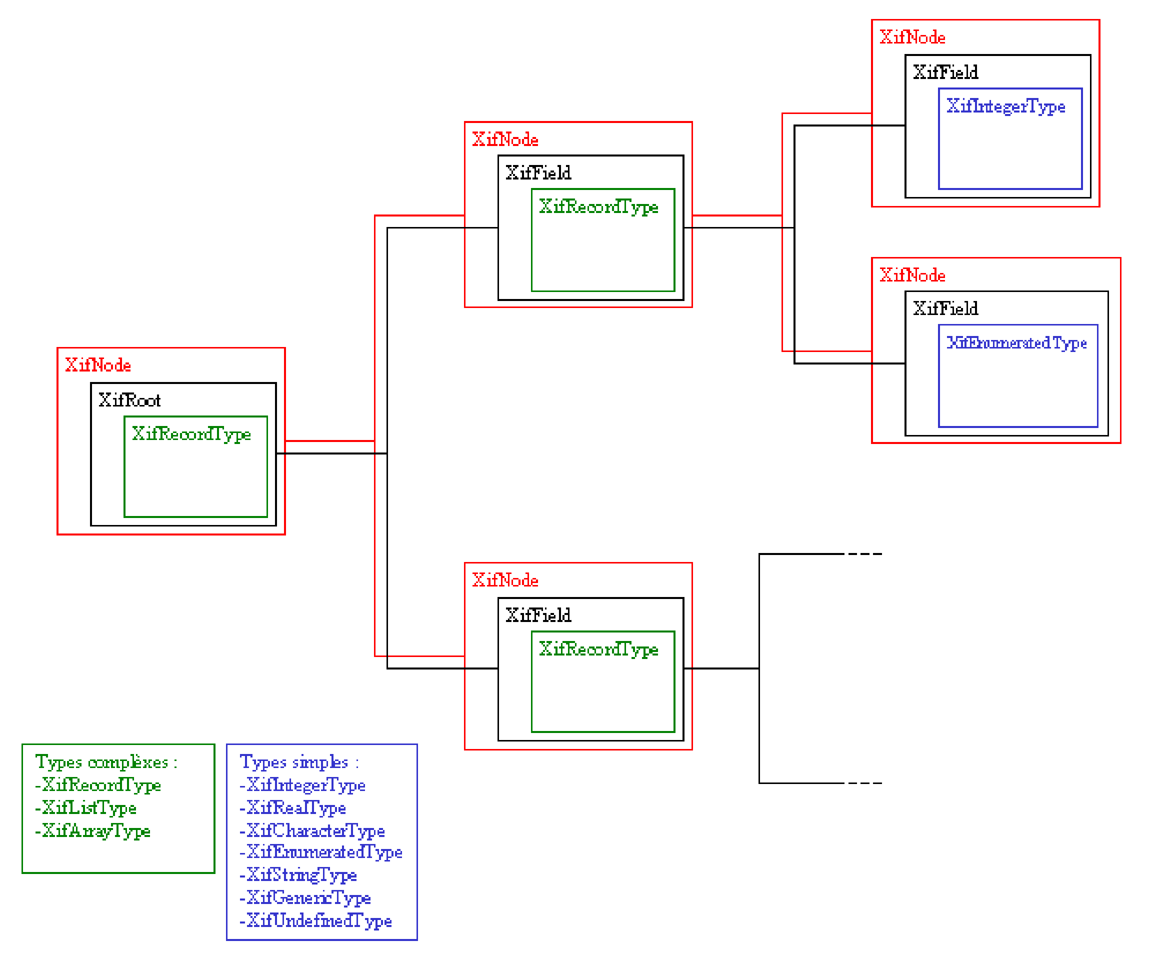


Figure 10 Structure de l'arbre XIF en mémoire dans l'API OASIS

Comme on peut le voir dans le schéma précédent, la représentation en mémoire de l'arbre XIF s'appuie sur une encapsulation d'objet. Chaque objet « XifNode » encapsule l'élément « XifField » (ou « XifRoot » si celui-ci est la racine de l'arbre) contenant les informations du champ (ou paramètre) et portant le type de celui-ci.

Il est donc fortement recommandé de s'appuyer sur les objets « XifNode » pour effectuer le parcours de l'arbre XIF étant donné que ces éléments portent l'ensemble des informations nécessaires à la manipulation de l'arbre. De plus, certaines manipulations telles que celles des conditions d'existences nécessitent de manipuler les objets « XifNode ».

Le parcours peut être réalisé de la manière suivante :

```
private void parseNode(XifNode aChild) {
      // La manipulation des champs ou types s'appuie sur l'objet "XifObject" étant
      // donnée que chaque élément héritent de cette classe. Il est donc possible
      // d'effectuer un parcours générique au moyen de cet objet.
      XifObject field = aChild.getUserObject();
      if (field.isRoot()) {
            System.out.println("--- Processing root named = " + field.getName() + "---");
      } else {
            System.out.println("--- Processing field named = " + field.getName() + "---");
      }
```

```
        // Récupération du type du champ.
        XifObject type = field.getType();
        // Analyse du type et traitement spécifique en function du type.
        if (type.isArrayType()) {
                parseArray(aChild, (XifArrayType)type);
        }
        else if (type.isCharacterType()) {
                parseCharacter(aChild, (XifCharacterType)type);
        }
        else if (type.isEnumeratedType()) {
                parseEnumerated(aChild, (XifEnumeratedType)type);
        }
        else if (type.isIntegerType()) {
                parseInteger(aChild, (XifIntegerType)type);
        }
        else if (type.isListType()) {
                parseNode(aChild.getChildNodes().get(0));
        }
        else if (type.isRealType()) {
                parseReal(aChild, (XifRealType)type);
        }
        else if (type.isRecordType()) {
                parseRecord(aChild, (XifRecordType)type);
        }
        else if (type.isStringType()) {
                parseString(aChild, (XifStringType)type);
        }
        else if (type.isUndefinedType()) {
                parseUndefined(aChild, (XifUndefinedType)type);
        }
        else if (type.isGenericType()) {
                parseGeneric(aChild, (XifGenericType)type);
        }
        parseDocumentation(aChild, type);
        parseExtension(aChild, type);
}
```

Chaque méthode appelée ci-dessous prend en paramètre le type à traiter mais également l'objet « XifNode ». Comme nous l'avons précisé précédemment, l'objet « XifNode » embarquant l'ensemble des informations du nœud et facilitant le parcours de l'arbre, il est important de le transmettre aux méthodes de traitement pour effectuer d'éventuelles manipulations sur l'arbre mais également pour utiliser l'ensemble des informations embarquées dans le nœud pour effectuer un affichage IHM par exemple.

Le parcours peut également être réalisé de la manière suivante :

```
private void parseNode(XifNode aChild) {
        // La manipulation des champs ou types s'appuie sur l'objet "XifObject" étant
        // donnée que chaque élément hérite de cette classe.
        XifObject field = aChild.getUserObject();
        if (field.isRoot()) {
                System.out.println("root named = " + field.getName());
        } else {
                System.out.println("field named = " + field.getName());
        }

        // Traitement spécifique sur le type
        // Récupération du type du champ.
        XifObject type = field.getType();
```

```
...

      // Traitement spécifique sémantique (field et type)
      parseDocumentation(field);
      parseDocumentation(type);

      // Traitement spécifique extension
      parseExtension(aChild);

      // Parcours récursif sur les fils si type complexe
      int nbChildren = aChild.getChildNodeCount();
            // process all children
            for (int i = 0 ; i < nbChildren ; i++) {
                  // gets child to be processed
                  XifNode child = aChild.getChildNodeAt(i);
                  parseNode(child);
      }
}
```

Cet exemple propose un parcours récursif se basant sur les objets « XifNode » sans se préoccupper du type de l'objet. Les traitements spécifiques au type sont réalisés indépendament du parcours de l'arbre.

7.1.1.4. Parcours d'un élément de type entier

L'exemple illustré ci-dessous montre le parcours d'un élément de type entier pour la récupération des valeurs qui lui sont associées et pouvoir les réutiliser dans le cadre d'un développement particulier. Chaque type peut ainsi être parcouru avec les méthodes spécifiques à chaque type qui sont détaillées dans la documentation JAVA de l'API OASIS.

Un certain nombre d'autres méthodes permettent de récupérer les détails du type entier et l'exemple ci-dessous n'offre pas une vue complète de ces possibilités.

```
private void parseInteger(XifNode aNode, XifIntegerType anInteger) {


      // range mode
      if (anInteger.isBeginEndRange()) {
            // Begin End range mode.
      }
      else if (anInteger.isFullRange()){
            // Full range mode.
      }
      else {
            // Begin Size range mode.
      }
      BigInteger beginValue = anInteger.getRange().getBegin();
      BigInteger endValue = anInteger.getRange().getEnd();
      BigInteger sizeValue = anInteger.getRange().getSize();

      // integer size
      long bitsSize = anInteger.getBitsSize();

      // encoding
      if (anInteger.isAscii()) {
            // ASCII mode.
      }
      else {
            // BINARY mode.
```

```
        }
}
```

7.1.1.5. Parcours d'un élément de type record

L'exemple illustré ci-dessous montre le parcours d'un élément de type record. Le passage de l'objet « XifNode » a un intérêt particulier ici puisque le parcours de l'arbre doit être poursuivi dans les éléments fils de ce type.

Comme on peut le voir dans le parcours, on effectue un appel à la méthode « parseNode(XifNode aChild) » qui constitue un appel récursif pour parcourir l'arbre jusqu'aux feuilles représentées par les types simples du format XIF.

```
private void parseRecord(XifNode aNode, XifRecordType aRecord) {
        int nbChildren = aRecord.getNumberOfFields();
        for (int i = 0 ; i < nbChildren ; i++) {
                XifNode child = aNode.getChildNodeAt(i);
                // traitement des elements fils.
                parseNode(child);
        }
        // Traitement spécifique sur le type record
        // ...

}
```

7.1.1.6. Parcours de la documentation sémantique d'un élément

La documentation se présente sous la forme d'une liste de paramètres ayant au moins un nom et une valeur. L'outil OASIS Modeller gère une liste par défaut d'attributs sémantiques qui sont définis dans le standard DEDSL.

L'API OASIS ne gère pas cette liste par défaut d'attributs sémantiques mais offre les outils pour les lire. Il est ensuite possible d'en écrire si l'on connaît la sémantique du standard DEDSL ou d'en écrire d'autres spécifiques à une définition.

L'exemple ci-dessous illustre le parcours des attributs sémantiques d'un élément passé en paramètres de la méthode JAVA.

```
private void parseDocumentation(XifObject xifObject) {
        XifDocumentation xifDoc = (XifDocumentation) xifObject.getDocumentation();
        if (xifDoc != null) {
                for (int i = 0; i < xifDoc.getNumberOfDocumentationParameters(); i++) {
                        XifDocumentationParameter param =
xifDoc.getDocumentationParameterAt(i);
                        String semanticAttributeName    = param.getName();
                        String semanticAttributeValue   = param.getValue();
                        String semanticAttributeContext = param.getContext();
                }
        }
}
```

7.1.1.7. Parcours des extensions d'un élément

Le format XIF offre un mécanisme qui permet de stocker les informations spécifiques à chacun des formats de

sortie sans tenir compte de la sémantique de celles-ci. Ce mécanisme se veut donc générique et extensible à volonté de façon à avoir un comportement homogène, quelque soit le type d'information à stocker ou le format cible.

Le format XIF offre donc des balises nommées « extensions » permettant d'ajouter un ensemble d'informations liées à un élément de l'arbre XIF qu'il soit « root », « field » ou « type ».

Afin de piloter et contrôler le contenu des extensions, un schéma XML des extensions (DR2) est associé au format XIF afin de définir la description syntaxique de chacune d'elles.

L'exemple ci-dessous illustre un parcours des extensions d'un élément sans distinction de son contenu. On ne se soucie donc pas dans cet exemple de récupération d'une information pour une extension spécifique mais on en récupère l'ensemble pour le traiter :

```
private void parseExtension(XifNode aNode, XifObject xifObject) {
	XifExtension xifExtension = (XifExtension) xifObject.getExtension();
	if ((xifExtension != null) && (xifExtension.getNumberOfParameters() > 0)) {
		Node anExtensionNode = xifExtension.getParameterAt(0);
		if (anExtensionNode instanceof Element) {
			parseExtensionNode(anExtensionNode);
		}
	}
}

private void parseExtensionNode(Node aNode) {
	String nodeName = aNode.getNodeName();
	String nodeValue = aNode.getNodeValue();
	NamedNodeMap map = aNode.getAttributes();
	if ((map != null) &&(map.getLength() > 0)) {
		for (int i = 0; i < map.getLength(); i++) {
			Node attribute = map.item(i);
			String attributeName = attribute.getNodeName();
			String attributeValue = attribute.getNodeValue();
		}
	}
	for (int i = 0; i < aNode.getChildNodes().getLength(); i++) {
		Node child = aNode.getChildNodes().item(i);
		if (child instanceof Element) {
			parseExtensionNode(child);
		}
	}
}
```

Rappel : l'exemple présenté ci-dessus illustre le parcours générique des extensions en s'appuyant sur les objets fournis par l'API DOM. La définition des éléments des extensions étant méconnue de l'API OASIS, le parcours de ces éléments ne peut pas être réalisé via les objets JAVA fournis par cette même API, c'est la raison pour laquelle l'exemple ci-dessus s'appuie sur un parcours via l'API DOM.

7.1.1.8. Récupération des informations d'un « calibrator »

Un certain nombre d'extensions définies pour le domaine du « Monitoring and Control » des télémesures et des télécommandes sont « factorisées » dans les fichiers XIF. On parle de factorisation car elles sont définies dans l'élément « generalInformation » du descriptif XIF et sont réutilisables une ou plusieurs fois dans les éléments de l'arbre XIF au moyen du « référencement » de ceux-ci. Ces références sont caractérisées par l'attribut « ref » dans les extensions qui le permettent.

Ce mécanisme a pour avantage de définir une extension de manière générale et de pouvoir la réutiliser autant de

fois souhaité dans les éléments de l'arbre. Le second avantage est que lors de la modification de cette extension, l'ensemble des éléments qui la référence est impacté par la modification et il n'est pas nécessaire de parcourir l'ensemble des éléments de l'arbre pour les modifier un à un.

Ce mécanisme s'applique pour les extensions nommées « calibrator ».

L'exemple suivant montre la manière de récupérer la liste des noms des « calibrator » définis de manière générale et qui peuvent être réutilisés dans l'arbre.

La liste des fonctions de transfert de type « calibrator » étant définies de manière globale dans le noeud « generalInformation » du modèle XIF, l'accès à ce nœud se fait par la classe « XifDataModel » portée par la classe « XifResource », c'est la raison pour laquelle la méthode présentée en exemple ci-dessous prend l'instance de cette classe en paramètre pour accéder à ce nœud via le modèle de données et ainsi récupérer la liste des fonctions de transfert qui y sont définies.

```
public static ArrayList<String> getCalibratorNameList(XifResource anXifResource) {
  ArrayList<String> nameList = new ArrayList<String>();
  XifObject generalInformation = anXifResource.getXifDataModel().getGeneralInformation();
  //get the calibrator list
  Node decalibratorListNode = XifExtensionUtilities.getExtensionFor(generalInformation,
"CALIBRATORLIST");
  if (decalibratorListNode != null) {
    for (Node decalibrator : XifExtensionUtilities.getAllChildrenNamed(decalibratorListNode,
"CALIBRATOR")) {
        String attributeValue = XifExtensionUtilities.getValueAttribute(decalibrator, "name");
        nameList.add(attributeValue);
    }
  }
  return nameList;
}
```

Une fois cette liste récupérée, sur chaque élément de l'arbre ayant une extension de ce type :

<DEFAULTCALIBRATOR calibratorRef="PID"/>

il est possible de vérifier si la définition du « calibrator » existe bien ou d'en récupérer la définition complète pour pouvoir le traiter.

7.1.2. Lecture d'un fichier XIF référençant d'autres fichiers XIF et récupération des données

Le format XIF offre la possibilité de référencer des fichiers entre eux, c'est-à-dire qu'un fichier XIF peut faire référence à d'autres fichiers XIF et ainsi utiliser les constantes, les types internes ou encore les champs globaux définis dans ceux-ci.

Recommandation : l'API OASIS n'interdit et n'empêche pas de modifier des éléments d'un XIF référencé par le XIF « maître » chargé en mémoire dans le modèle de données porté par la classe « XifResource ». Il est cependant fortement déconseillé de modifier quoique ce soit dans les fichiers XIF qui peuvent être référencés. En effet, la modification d'un élément d'un fichier référencé peut avoir des impacts non contrôlés sur d'autres fichiers référençant ce même fichier. Il est donc préférable de ne modifier que les éléments du XIF « maître » chargé en mémoire et de n'effectuer que des références sur les éléments des fichiers XIF référencés.

Dans le cas où l'on souhaite volontairement modifier un élément d'un XIF référencé, il est préférable de

charger ce fichier XIF en tant que XIF maître dans une nouvelle ressource « XifResource » et d'y effectuer les modifications souhaitées.

Le référencement se caractérise par la balise « XifReferenceList » contenant un ensemble de balise « XifReference » pour chaque fichier XIF référencé.

Pour renseigner cette balise, se référer au chapitre 7.2.1.3.

Pour lire un fichier XIF contenant des références vers un autre fichier XIF, il n'y a rien de spécifique à réaliser étant donné que la classe « XifResource » gère le chargement du fichier XIF principal ainsi que de ses fichiers XIF référencés.

7.1.3. Lecture de plusieurs fichiers XIF en parallèle

Il est possible de charger et de gérer plusieurs fichiers XIF en parallèle sans interférer la manipulation des modèles de données en mémoire.

La classe « XifDataManager » du paquetage « modeller.core.dataManager » permet de gérer plusieurs instances de la classe « XifResource » en mémoire et de manipuler ces modèles de données de manière indépendantes.

Voir la javadoc de l'API OASIS pour plus d'informations sur cette classe.

7.2. ECRITURE D'UN FICHIER XIF

7.2.1. Création d'un modèle en mode « Monitoring And Control » et écriture du fichier XIF

Ce chapitre montre les clés principales pour créer un fichier XIF en s'appuyant sur la création d'un exemple simple du mode « Monitoring And Control ». Le but est donc de créer un XIF comme suit :

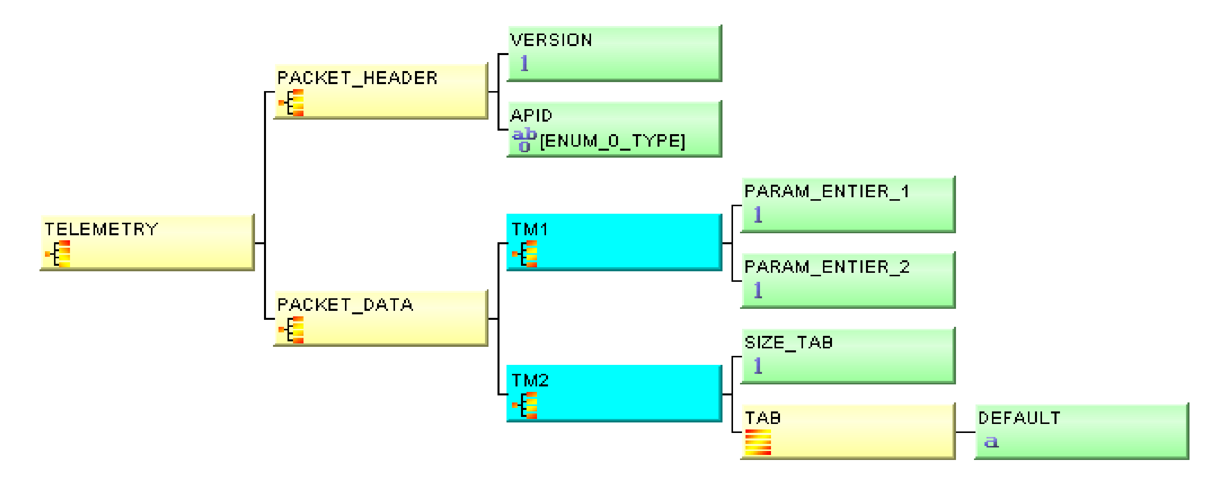


Figure 11 Représentation graphique du nœud « TELEMETRY »

Cet exemple modélise un arbre de TM ayant pour spécification :

- Une en-tête « PACKET_HEADER » contenant un élément de type entier « VERSION » et un APID de type énuméré.
- Une donnée « PACKET_DATA » contenant un élément de type record modélisant :
  - Un premier élément « TM1 » contenant deux entiers et discriminé par la valeur « UN » de l'APID.
  - Un second élément « TM2 » contenant un entier et un tableau de caractères dont la taille dépend de la valeur de l'entier de même niveau.
- Les paramètres « PARAM_ENTIER_1 » et « PARAM_ENTIER_2 » sont respectivement affectés de deux extensions: le premier avec une alarme et le second avec une fonction de « calibration ».

Les éléments de couleur « Jaune » dans l'arbre sont les éléments de type « complexe », c'est-à-dire pouvant contenir d'autres éléments. On dénombre 3 types complexes en XIF : XifRecordType, XifArrayType et XifListType.

Les éléments de couleur « Verte » dans l'arbre sont les éléments de type « simple », c'est-à-dire les feuilles de l'arbre XIF. On dénombre 5 types simples en XIF : XifIntegerType, XifRealType, XifCharacterType, XifEnumeratedType et XifStringType.

Il existe deux autres types « spécifiques » que sont XifUndefinedType et XifGenericType.

Les éléments de couleur « Bleue » dans l'arbre sont les éléments dont l'existence est conditionnée par la valeur d'un autre élément.

Au final, le XIF résultant de cette modélisation est présenté en Annexe B de ce document.

Les sources des classes JAVA permettant de modéliser ce descriptif sont présentées en Annexe C de ce document.

7.2.1.1. Initialisation de la configuration du modeller

Deux points sont essentiels pour le fonctionnement nominal de l'API OASIS :

Le premier est de spécifier la variable globale « best.home » à la Machine Virtuelle Java qui exécute le programme faisant référence à l'API OASIS. Pour cela, il est nécessaire de passer en paramètre de l'appel « java » cette variable sous la forme :

-Dbest.home=api_oasis/exe *où "api_oasis/exe" est le chemin relatif vers le répertoire "exe" contenant le jar "modeller.jar" de l'API OASIS, à partir du répertoire d'exécution du programme.*

Le second est d'initialiser la configuration de l'API au tout début du programme développé dans le langage JAVA. Pour cela, la méthode d'appel est la suivante :

```
// Chargement de la configuration du Modeller.
ModellerConfiguration.loadModellerConfiguration();
```

Cette méthode permet notamment de récupérer les informations telles que :

- la version du modeller utilisée,
- les informations de configuration des machines définies dans une configuration spécifique sous « exe\modeller\configuration\configurationFiles\squelettes\BINREPS » ,

- le chemin vers le schéma XML des extensions,
- les éventuels « plugins » embarqués.

Ces deux étapes sont donc essentielles pour assurer le bon fonctionnement de l'API OASIS et sont donc les premiers éléments à vérifier lorsqu'une erreur apparaît dans les premiers développements d'une application s'appuyant sur l'API OASIS.

Il est à noter que lors du chargement d'un fichier XIF via l'interface « ModellerServicesInterface », le chargement de la configuration est réalisé automatiquement. Il n'est donc pas nécessaire, uniquement dans ce cas précis, de faire appel à cette méthode.

7.2.1.2. Initialisation des objets porteurs du modèle de données en mémoire

Les 3 classes principales de manipulation du modèle de données d'un descriptif XIF appartiennent toutes au paquetage « `modeller.core.dataManager.xifManager` » et sont les suivantes :

- « `XifResource` »,
- « `XifDataModel` »,
- « `XifDataModelHandler` ».

La classe principale portant l'ensemble du modèle de données chargé en mémoire et correspondant à une description XIF est la classe « `XifResource` ».

Cette classe constitue **le point d'entrée** de la manipulation du modèle de données chargé en mémoire. Elle fournit un certain nombre de méthode de haut niveau de manipulation du modèle de données telles que :

- la création d'un nouveau modèle,
- le chargement d'un modèle,
- La sauvegarde d'un modèle,
- Le chargement de sous modèle correspondant aux références d'un fichier XIF vers d'autres « sous-XIF »,
- Le rattachement ou le détachement d'un sous modèle correspondant à la référence vers un « sous-XIF »,
- Etc.…

Le schéma ci dessous illustre la correspondance entre plusieurs dépendances entre fichiers XIF et le modèle de données correspondant en mémoire, « encapsulé » par l'instance de « XifResource ». Tout modèle, principal, référencé directement ou indirectement par le modèle principal est une instance de « XifDataModel » ayant accès via le lien « xifResource » à la resource qu'il compose.

Si un même fichier XIF est référencé plusieurs fois, c'est la même instance de « XifDataModel » qui est liée à l'objet référençant, comme par exemple pour le fichier « file4.xif » référencé par « file2.xif » et « file3.xif ». Sur le même principe l'instance correspondant à « file1.xif » est utilisé en tant que modèle principal par la resource mais est aussi réutilisé par le référencement depuis « file4.xif ».

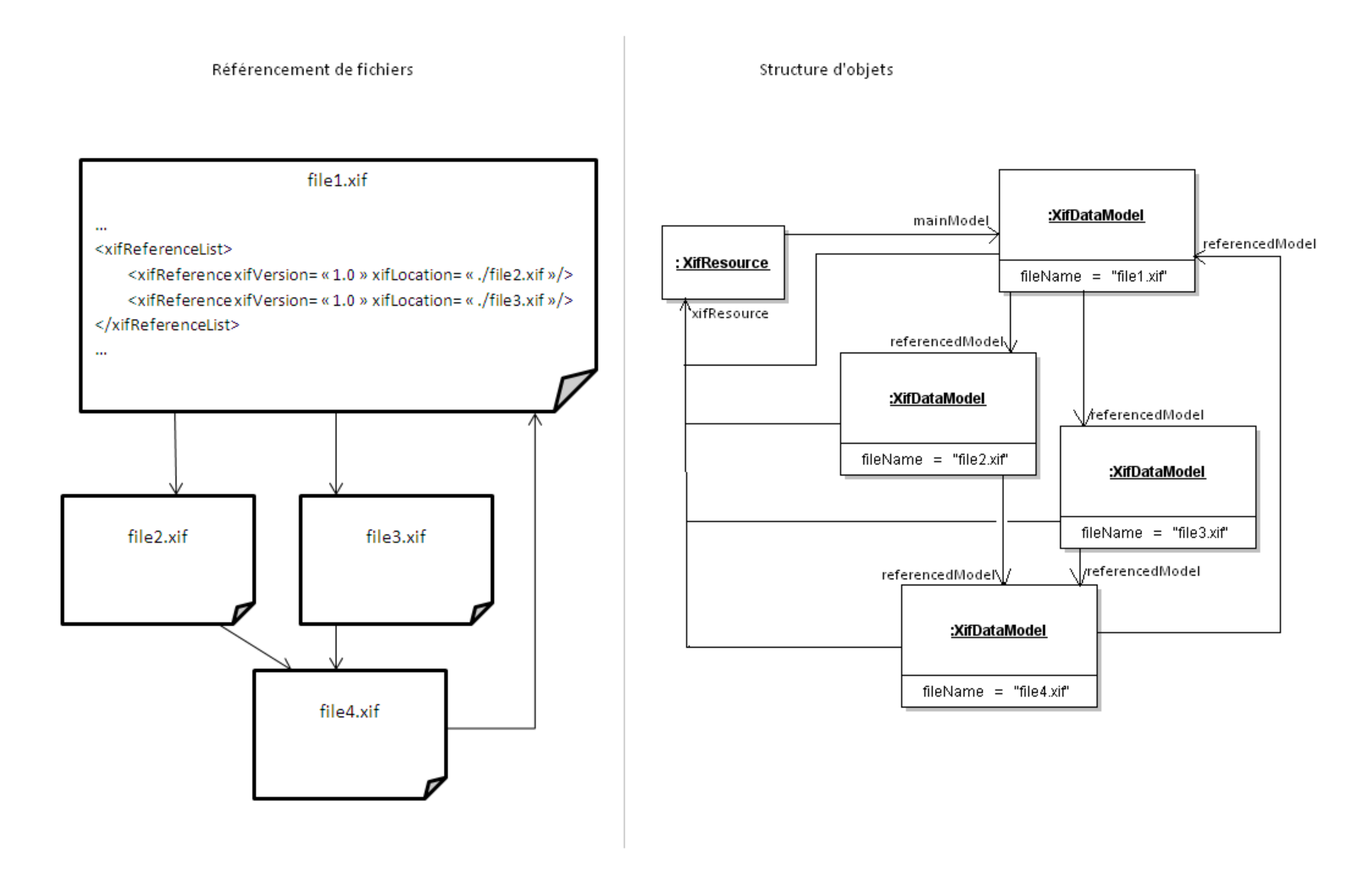


Figure 12 Mapping du référencement de fichiers XIF et de la structure en mémoire

La classe « XifDataModelHandler » est accessible depuis le modèle de données du XIF principal. Elle offre l'ensemble des méthodes d'ajout, de modification et de suppression des éléments globaux ou locaux du descriptif XIF.

L'exemple de code suivant illustre l'initialisation de ces 3 classes centrales pour l'exemple que nous allons illustrer au cours de ce chapitre mais également pour toute utilisation de l'API OASIS destiné à créer et manipuler un modèle de données XIF.

```
// On créé un nouveau XifResource et on lui associe un nouveau XifDataModel.
XifResource xifResource = new XifResource();

// Le XifDataModel est créé avec un nom de XIF qui correspond au nom du fichier XIF
et un élément root
// est créé par défaut avec ce nom. Le mode "EAST", "M_C" ou encore "XSD" est passé
également
// en paramètre pour spécifier le mode du XIF si celui-ci doit être ouvert dans OASIS
Modeller.
xifResource.createNewDataModel(XIF_NAME, "M_C");

// On récupère le XifDataModel qui est unique dans ce cas et qui correspond au XIF en
mémoire.
XifDataModel mainDataModel = xifResource.getXifDataModel();
```

```
// On récupère le XifDataModelHandler qui fournit un bon nombre
// de méthode de création, modification de noeud dans l'arbre XIF.
XifDataModelHandler mainDataModelHandler = mainDataModel.getDataModelHandler();
```

La méthode « **public void** `createNewDataModel(String aFileName, String aMode)` » de la classe « XifResource » permet d'initialiser les principales informations générales du modèle XIF et notamment les différentes versions ainsi que le mode qui lui est passé en paramètre. Dans l'exemple que nous nous attachons à construire dans ce chapitre, le mode utilisé est le mode « M_C » pour « Monitoring And Control ».

Cette méthode n'initialise cependant pas l'ensemble des informations générales et il est nécessaire d'en traiter certaines comme le montre le paragraphe suivant.

7.2.1.3. Initialisation des informations générales du XIF

Comme le paragraphe précédant le montre, la méthode « createNewDataModel » de la classe « XifResource » initialise les éléments connus de la configuration de l'API OASIS et notamment les différentes versions définies dans le nœud « generalInformation ».

Dans le cas où l'on souhaite spécifier à l'API OASIS où sauvegarder le XIF à la fin de la création du modèle, celui-ci peut porter le chemin vers le répertoire où sauvegarder le fichier XIF résultat avec l'exemple ci-dessous.

```
// On lui affecte un répertoire dans lequel sera sauvegardé le XIF.
mainDataModel.setDirectory("D:\\MON_PROJET_BEST\\");
```

Les informations de la balise « targetNamespace » peuvent être renseignées pour l'identification du modèle. L'affectation d'un préfixe permet d'éviter les ambiguïtés de référencement d'objets de même nom dans différents modèles externes.

Le code suivant montre la manière de renseigner cette balise :

```
// Affectation du targetNameSpace
mainDataModel.setTargetNamespace("http://myNameSpace/myXif", "myxif");
```

7.2.1.4. Initialisation des informations physiques

A la création du modèle de données, il est possible de définir les informations d'encodage des données que le fichier XIF décrit. Pour cela, il est possible de définir les informations appelées physiques de la manière suivante :

```
// Ajout des informations physiques au noeud general information.
XifPhysicalInformation physicalInfo = mainDataModel.getPhysicalInformation();
physicalInfo.setNorm("FCSTC000");
physicalInfo.setSendingMachine("PC()");
physicalInfo.setWordSize("16");
physicalInfo.setArrayArranging("ROW");
```

Ces informations sont composées de 4 attributs : la norme, la machine, la taille d'un mot et la disposition des informations en mémoire. Le document de référence DR1 explique plus en détail les différentes combinaisons possibles pour initialiser ces informations physiques liées à la machine.

Cette définition suit les conventions régies par le CCSDS et est configurée dans l'API sous le paquetage « modeller.configuration.configurationFiles.squelettes ».

La liste des machines pré-configurées est la suivante :

- MACHINE01=PC()
- MACHINE02=SUN()
- MACHINE03=VAX()
- MACHINE04=MIL_STD_1750A()
- MACHINE05=CDC()
- MACHINE06=IBM(CMS-MVS)

7.2.1.5. Création du ou des nœuds « ROOT »

L'exemple que nous suivons s'appuie sur le mode « Monitoring And Control » qui, par défaut, doit contenir trois nœuds « ROOT » correspondant à trois descriptions arborescentes distinctes.

Ces trois nœuds correspondent aux définitions :

- du « SPACE SYSTEM » et des équipements qui le composent,
- de la « TELEMETRY » et de la définition du paquet TM correspondant,
- de la « COMMAND » et de la définition du paquet TC correspondant.

Il est donc nécessaire de créer ces 3 nœuds « ROOT » de la manière suivante :

```
// On supprime le noeud root créé par défaut.
mainDataModelHandler.removeRoot(mainDataModel.getRootAt(0));

// Création et ajout du noeud root SPACE_SYSTEM à l'arbre XIF
//Ajout de l'extension <ROOTNATURE>spaceSystem</ROOTNATURE> permettant
// de definir ce noeud root comme etant le noeud "SPACE SYSTEM"
XifNode spaceSystemRootNode = mainDataModelHandler.addNewRoot("SPACE_SYSTEM");
XifExtensionUtilities.setExtensionValueFor(spaceSystemRootNode.getUserObject(),
rootNature, "spaceSystem");

// Création et ajout du noeud root TELEMETRY à l'arbre XIF
//Ajout de l'extension <ROOTNATURE>telemetry</ROOTNATURE> permettant
// de definir ce noeud root comme etant le noeud "TELEMETRY"
XifNode telemetryNode = mainDataModelHandler.addNewRoot("TELEMETRY");
XifExtensionUtilities.setExtensionValueFor(telemetryNode.getUserObject(),
rootNature, "telemetry");

// Création et ajout du noeud root COMMAND à l'arbre XIF
//Ajout de l'extension <ROOTNATURE>command</ROOTNATURE> permettant
// de definir ce noeud root comme etant le noeud "COMMAND"
XifNode commandNode = mainDataModelHandler.addNewRoot("COMMAND");
XifExtensionUtilities.setExtensionValueFor(commandNode.getUserObject(),
rootNature, "command");
```

Comme nous l'avons vu précédemment, la création du squelette du modèle de données en mémoire est initialisé par l'appel « `xifResource.createNewDataModel(XIF_NAME, "M_C");` ». Cette méthode initialise en mémoire un nœud « root » par défaut nommé avec le nom du XIF lui-même. Il est donc nécessaire de supprimer

ce nœud « root » par défaut qui ne nous intéresse pas dans notre cas puis de créer les 3 nœuds « root » souhaités comme l'exemple le montre.

7.2.1.6. Création du paquet « PACKET_HEADER »

Le paragraphe suivant s'attache à illustrer la création de la branche « PACKET_HEADER » de l'exemple suivi dans ce chapitre :

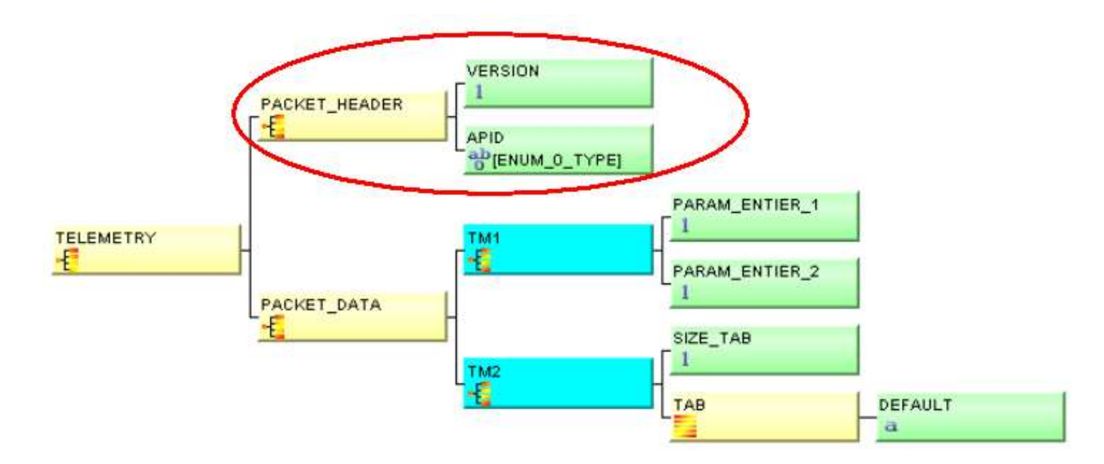


Figure 10 Représentation graphique du nœud « PACKET_HEADER »

Dans un premier temps, on s'attache à créer le nœud « PACKET_HEADER » de cette branche pour ensuite, dans un second temps, y attacher les paramètres qu'il contient.

La création d'un élément de l'arbre s'effectue en 4 étapes :

- La création du type constituant l'élément,
- La création du champ « field » constituant l'élément,
- Le rattachement du type au champ de l'élément,
- Le rattachement du champ à l'arbre XIF.

Une cinquième étape annexe permet de récupérer le nœud « XifNode » de l'élément créé pour une utilisation ultérieure.

Le code suivant illustre la création du champ « PACKET_HEADER ».

Ce code est ensuite détaillé dans les chapitres suivants.

```
// 1 - Création du field.
```

```
XifField fieldHeader = XifField.createDefaultXifField("PACKET_HEADER",
mainDataModel);

// 2 - Création du type
XifRecordType header = XifRecordType.createDefaultXifRecordType(mainDataModel);

// 3 - rattachement du type au field.
fieldHeader.setType(header);

// 4 - Ajout du field dans le modèle.
mainDataModelHandler.addElement(rootNode, fieldHeader);

// 5 - Récupération du noeud créé.
XifNode headerNode = rootNode.getChildNodeNamed("PACKET_HEADER");
```

7.2.1.6.1. Création d’un type

Les objets « JAVA » correspondants à chaque type du modèle XIF sont contenus dans le paquetage :

```
package modeller.core.dataManager.xifManager.api;
```

Comme cela est décrit dans le paragraphe 4.3, ces classes surchargent les classes générées par l’API JAXB et basées sur la définition XSD du format XIF.

L’ensemble des types possibles est représenté par le sigle « XXX » dans la suite de ce paragraphe.

Il existe donc autant de classe « XifXXXType » que de type définit par la définition XSD du format XIF.

Chaque classe « XifXXXType » correspondant à chaque type contient deux méthodes permettant d’initialiser une instance de ce type :

```
public static XifXXXType createDefaultXifXXXType(XifDataModel aModel)
```

et

```
public static XifXXXType createReferenceForXifXXXType(XifXXXType aType, XifDataModel
aModel)
```

Il est fortement recommandé d’utiliser ces méthodes pour la création d’un nouveau type et déconseillé d’utiliser l’instanciation par le constructeur de la classe tel que :

```
XifXXXType aType = new XifXXXType() ;
```

En effet, ces méthodes statiques assurent une initialisation minimale de chaque type afin de minimiser les erreurs dans leur manipulation et leur attachement à l’arbre XIF.

La première méthode permet de créer un type « XXX » en lui affectant le modèle de données « XifDataModel » de destination et en lui affectant un certain nombre d’attributs par défaut en fonction du type choisi. Une fois le type souhaité retourné par la méthode, il suffit de lui affecter les attributs souhaités.

La seconde méthode permet de créer une référence sur le type (de même type) qui lui est passé en paramètre. Cette méthode est utilisée pour référencer un type interne définit dans la liste des types internes du XIF (nœud

« typeList »). Nous verrons un exemple d'utilisation de cette seconde méthode dans la création d'une référence sur un type dans le chapitre 7.2.1.7.2.

7.2.1.6.2. Création du type « XifRecordType » pour le champ « PACKET_HEADER »

Comme le précise le paragraphe précédent, il est nécessaire de créer une instance de type « XifRecordType » à partir de la méthode statique fournit par cette même classe.

Le champ « PACKET_HEADER » est nommé mais contient un type local et ne référence pas un type interne, le type record créé ici ne doit donc pas être nommé.

Si l'on reprend la première étape de création du champ « PACKET_HEADER », la création du type « XifRecordType » était réalisée de la manière suivante :

```
XifRecordType header = XifRecordType.createDefaultXifRecordType(mainDataModel);
```

Comme on le précise précédemment, on utilise la méthode « createDefaultXifXXXType(XifDataModel aModel) » du type « XifRecordType » pour créer l'instance du type. On peut ensuite lui affecter un nom. Ici, on ne souhaite pas qu'il soit nommé, la méthode initialise à « null » le nom du type retourné.

7.2.1.6.3. Création du champ « XifField »

La seconde étape est de créer le champ « XifField » qui sera affecté du type « XifRecordType » créé précédemment.

Ce champ est nommé « PACKET_HEADER », donc l'appel est réalisé sous la forme suivante :

```
XifField fieldHeader = XifField.createDefaultXifField("PACKET_HEADER",
mainDataModel);
fieldHeader.setType(header);
```

La classe « XifField » contient au même titre que les classes portant les types, une méthode permettant d'instancier et d'initialiser l'objet souhaité. Cette méthode affecte le nom passé en paramètre à l'instance retournée ainsi qu'un type « XifUndefinedType » par défaut. Cette méthode créé un type « XifUndefinedType » par défaut attaché à ce champ.

Lorsque l'on affecte le type « XifRecordType » au champ avec la méthode « setType », ce type écrase et remplace le type créé par défaut.

7.2.1.6.4. Ajout du champ « PACKET_HEADER » à l'arbre XIF en cours de création

La classe « XifDataModelHandler » offre un bon nombre de méthode d'ajout de suppression ou de modification d'élément au cours de la manipulation d'un modèle de données XIF. Ces méthodes ont été conçues de manière à effectuer un certain nombre de vérification au cours de ces manipulations et d'assurer la cohérence du modèle de données pour chaque action réalisée. Il est donc fortement conseillé d'utiliser les méthodes de cette classe au cours de la manipulation des éléments dans l'arbre.

Pour ajouter le champ précédemment créé à l'arbre XIF en cours de création, il suffit d'écrire le code suivant :

```
mainDataModelHandler.addElement(rootNode, fieldHeader);
```

L'instance "mainDataModelHandler" de la classe « XifDataModelHandler » offre la méthode « addElement » permettant d'ajouter le champ « fieldHeader » au nœud « rootNode » de l'arbre en cours de création.

7.2.1.7. Création des paramètres du paquet « PACKET_HEADER »

Après avoir créé le champ « PACKET_HEADER », il est nécessaire de créer et d'ajouter les paramètres qui le composent.

7.2.1.7.1. Création du paramètre « VERSION »

La création du paramètre « VERSION » se caractérise par le code suivant, que l'on détaille par la suite.

```
// 1 - Création du type Entier de la version.
// Création du type Entier de la version.
XifIntegerType version = XifIntegerType.createDefaultXifIntegerType(mainDataModel);
// Creation de l'intervalle du type entier.
XifIntegerRange newRange = version.getIntegerRange();
newRange.setMinValue(BigInteger.ZERO);
newRange.setMaxValue(BigInteger.valueOf(100));
// Affectation des attributs du type Entier.
version.setIntegerRange(newRange);
version.setBitSizeComputed();
version.setBinary();
// la methode getBitsSize() permet de calculer la taille en bits de l'entier
// en fonction des informations affectees precedement.
version.setSize(Long.toString(version.getBitsSize()));

// 2 - Création du field de la version
XifField fieldVersion = XifField.createDefaultXifField("VERSION", mainDataModel);
fieldVersion.setType(version);

// 3 - Ajout de l'extension "telemetred" sur le parametre pour specifier la nature de
celui-ci.
WriterExtensionUtilities.addDataSourceForParameter(fieldVersion, "telemetered");

// 4 - Ajout du parametre a la liste des parametres.
mainDataModelHandler.addGlobalField(fieldVersion);

// 5 - Creation du field referencant le parametre global pour le rattacher au HEADER.
XifField fieldRefVersion = XifField.createReferenceForXifField(fieldVersion,
mainDataModel);

// 6 - Ajout du paramètre référencé au champ "PACKET_HEADER"
XifNode headerNode = rootNode.getChildNodeNamed(fieldHeader.getName());
mainDataModelHandler.addElement(headerNode, fieldRefVersion);
```

1) La première étape est de créer le type « XifIntegerType » pour le champ « VERSION ».

Cette étape reprend le même processus que la création du type « XifRecordType » vu précédemment. C'est-à-dire que ce code consiste à instancier l'objet du type et affecter les valeurs des attributs qui le composent.

Il est à noter que dans le cas où l'on souhaiterait renseigner le nom du type pour l'affecter à la liste des types internes, il serait nécessaire de vérifier que ce nom n'existe pas déjà dans la liste des types internes pour éviter les doublons. Comme on l'a précisé précédemment, un type nommé appartient à la liste des types internes et par conséquent, il ne peut pas y avoir deux types de même nom.

Pour effectuer la vérification il faudrait ajouter le code suivant :

```
String integerTypeName = "UN_NOM";
if (integerTypeName != null) {
      XifObject t = dataModel.getXifResource().getInternalTypeNamed(integerTypeName);
```

```
      if (t != null) {
            throw new Exception("Creation du type impossible : Le type [" +
integerTypeName + "] existe deja.");
      }
}
```

2) La seconde étape consiste à créer le champ « XifField » qui sera ensuite rattaché à l'arbre XIF.

3) La troisième étape consiste à affecter une extension à ce champ permettant aux outils tels que « OASIS Modeller » de déterminer le type de paramètre de ce champ ou « dataSource ». Les différents types de paramètres existants sont :

- « argument » pour les paramètres de télécommande,
- « constant » pour les paramètres systèmes,
- « derived » pour les paramètres dérivés,
- « local » pour les paramètres locaux (rarement utilisés),
- « telemetered » pour les paramètres télé-mesurés.

Dans le cas où un paramètre est utilisé à la fois dans l'arbre « telemetry » et dans l'arbre « command », la convention veut que ce paramètre soit affecté comme « telemetered ».

Dans l'exemple de ce chapitre, nous traitons des télémesures, il est donc nécessaire d'affecter le type « telemetered » aux paramètres créés ici.

La déclaration de la méthode utilisée est la suivante :

```
public static void addDataSourceForParameter(XifObject xifObject, String
dataSourceValue)
throws Exception {
      try {
            XifExtensionUtilities.setExtensionValueFor(xifObject, null,
"PARAMETERPROPERTIES");
            Node paramProperties =  XifExtensionUtilities.getExtensionFor(xifObject,
"PARAMETERPROPERTIES");
            XifExtensionUtilities.createOrSetAttributeNamed(paramProperties,
"dataSource", dataSourceValue);
      } catch (Exception e) {
            throw new Exception(e.getMessage());
      }
}
```

Dans cette méthode, on utilise les méthodes statiques de la classe « XifExtensionUtilities » fournie par l'API OASIS.

On créé d'abord l'extension « PARAMETERPROPERTIES » qui permet de définir les propriétés d'un paramètre puis on lui affecte l'attribut qui porte le type du paramètre ou « dataSource ».

4) La quatrième étape consiste à ajouter le paramètre à la liste des paramètres du modèle de données XIF. Un paramètre, quelque soit son type, est un élément qui peut être réutilisé et donc factorisé. Les paramètres étant les champs « XifField », ils appartiennent tous à la liste « fieldList » du fichier XIF que l'on nomme également « globalField ».

Cette étape est réalisée via l'instance « mainDataModelHandler » de la classe « XifDataModelHandler » et sa méthode « `mainDataModelHandler.addGlobalField(fieldVersion);` ».

5) L'étape suivante consiste à créer une référence vers le paramètre factorisé pour attacher cette référence à l'arbre XIF. En effet, le champ précédemment créé est affecté à la liste des champs globaux et il est nécessaire de créer un nouveau champ référençant ce paramètre global qui sera l'instance de la « VERSION » dans l'arbre XIF.

Pour cela, la méthode suivante fournie par la classe « XifField » permet de créer une référence à partir d'un paramètre connu :

```
XifField fieldRefVersion = XifField.createReferenceForXifField(fieldVersion,
mainDataModel);
```

6) La dernière étape consiste à ajouter ce champ référence à l'arbre XIF et notamment au champ « PACKET_HEADER ».

On récupère d'abord le nœud de l'arbre correspondant au champ « PACKET_HEADER » :

```
XifNode headerNode = rootNode.getChildNodeNamed(fieldHeader.getName());
```

Puis, on lui affecte le nouveau champ référence via les méthodes de la classe « XifDataModelHandler » :

```
mainDataModelHandler.addElement(headerNode, fieldRefVersion);
```

7.2.1.7.2. Création du paramètre « APID »

La création du paramètre « APID » se caractérise par le code suivant, que l'on détaille par la suite.

```
// 1 - Création du type énuméré affecté d'un nom par défaut
XifEnumeratedType apid =
XifEnumeratedType.createDefaultXifEnumeratedType(mainDataModel);

// 2 – Ajout des valeurs énumérées "UN" et "DEUX".
try {
   XifEnumerationValue enumValue =
XifEnumerationValue.createDefaultXifEnumeratedValue(mainDataModel, "UN");
   enumValue.setValue("1");
   apid.addEnumeratedValue(value);

   enumValue = XifEnumerationValue.createDefaultXifEnumeratedValue(mainDataModel,
"DEUX");
   enumValue.setValue("2");
   apid.addEnumeratedValue(value);
...
} catch (Exception e) {
   throw new Exception(e.getMessage());
}

// 3 – Affectation des attributs de l'énuméré.
apid.setBitSizeComputed();
apid.setBinaryRepresentation();
apid.setAscii();

// 4 - Ajout du type APID a la liste des types internes.
mainDataModelHandler.addInternalType(apid);
```

```
XifEnumeratedType apidRefType =
XifEnumeratedType.createReferenceForXifEnumeratedType(apid, mainDataModel);
// Création du field de l'APID
XifField filedApid = XifField.createDefaultXifField("APID", mainDataModel);
filedApid.setType(apidRefType);
// Ajout de l'extension "telemetred" sur le parametre pour specifier la nature de
celui-ci.
WriterExtensionUtilities.addDataSourceForParameter(filedApid, "telemetered");
// Ajout du parametre a la liste des parametres.
mainDataModelHandler.addGlobalField(filedApid);
// Creation du field referencant le parametre global pour le rattacher au HEADER.
XifField fieldRefApid = XifField.createReferenceForXifField(filedApid,
mainDataModel);
mainDataModelHandler.addElement(headerNode, fieldRefApid);
```

Les étapes de création du paramètre « APID » présenté ci-dessus sont sensiblement les mêmes que pour la création du paramètre « VERSION ».

Les différences remarquables commentées dans le code ci-dessus sont :

- Le type énuméré doit être affecté des valeurs de l'énuméré. Ces valeurs sont composées chacune de deux informations, un nom logique qui permet d'identifier la valeur et une valeur physique correspondant à la valeur de l'énumération et qui peut être vide en fonction de l'encodage de l'énuméré.

- Le type énuméré doit être nommé et donc appartenir à la liste des types internes du descriptif XIF. C'est la raison pour laquelle on utilise la méthode « addInternalType » de la classe « XifDataModelhandler ».

7.2.1.8. Création du paquet « PACKET_DATA »

Le paragraphe suivant s'attache à illustrer la création de la branche « PACKET_DATA » de l'exemple suivi dans ce chapitre :

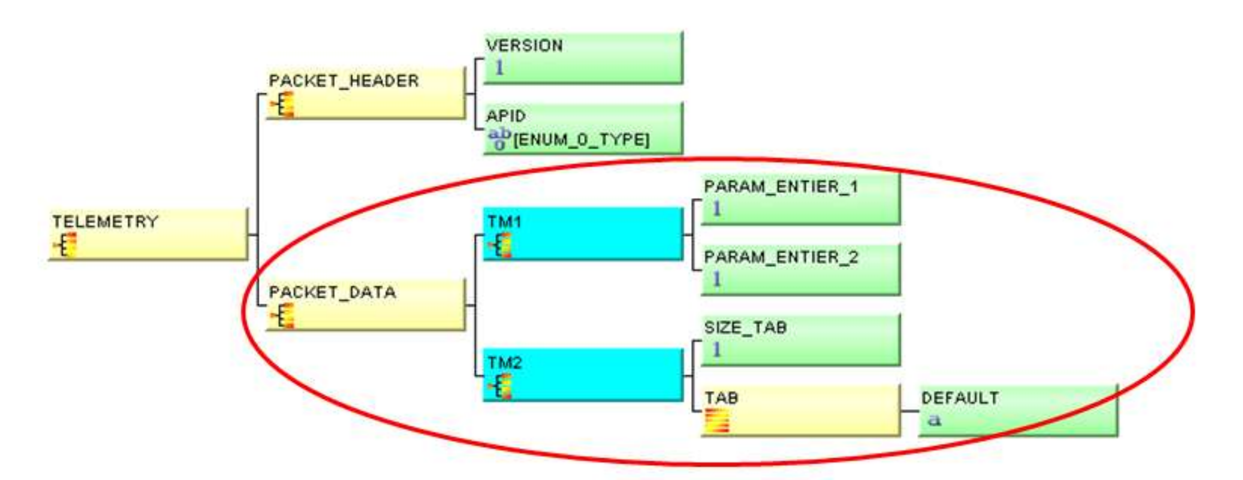


Figure 11 Représentation graphique du nœud « PACKET_DATA »

La création des différents éléments de ce paquet n'est pas détaillée dans ce chapitre étant donné qu'elle est redondante avec les étapes de création détaillées dans les chapitres précédents.

Les éléments présentés ci-après permettent de montrer :

- La manière d'affecter une condition d'existence à un élément de l'arbre XIF,
- La manière de créer une constante et de la référencer dans la définition de l'intervalle d'un entier,
- La manière d'affecter une fonction de transfert (calibration) à un paramètre,
- La manière d'affecter une surveillance (alarme) à un paramètre,
- La manière de définir une taille de tableau conditionné par la valeur d'un paramètre de l'arbre XIF.

7.2.1.8.1. Création de l'élément « TM1 »

Le code présenté ci-dessous montre la manière de définir un élément dans l'arbre et de lui affecter une condition d'existence en fonction de la valeur de l'énuméré « APID ».

```
// Creation du record TM1 et ajout a l'arbre.
XifRecordType tm1 = XifRecordType.createDefaultXifRecordType(mainDataModel);
XifField fieldtm1 = XifField.createDefaultXifField("TM1", mainDataModel);
fieldtm1.setType(tm1);
mainDataModelHandler.addElement(dataNode, fieldtm1);
XifNode aNode = rootNode.getChildNodeNamed(fieldtm1.getName());

// Création de la condition d'existence et rattachement au field
XifDiscriminantAssociation discAssos = new XifDiscriminantAssociation();
discAssos.createAssociationTable(fieldtm1, "TELEMETRY/PACKET_HEADER/APID[UN]",
fieldtm1.getDataModel());
fieldtm1.setConditionAssociation(discAssos);

// Création de l'arbre de conditions d'existence.
ConditionNode cond = ConditionManager.processFieldCondition(aNode);
aField.setExistenceCondition(cond);
```

Les 4 premières lignes du code ci-dessus reprennent la création de l'élément « TM1 » de la même manière que cela a été vu dans les chapitres précédents.

La suite montre la création d'une condition d'existence portée par la classe « XifDiscriminantAssociation » et dont la méthode permet de créer la condition en lui fournissant sous forme de chaîne de caractère l'élément discriminant.

Ici, l'élément discriminant est l'énuméré « APID » dont le chemin dans l'arbre est « TELEMETRY/PACKET_HEADER/APID ». La syntaxe « [UN] » permet de préciser la valeur de l'APID pour laquelle l'élément « TM1 » existe. Cela signifie donc que l'élément « TM1 » et son contenu n'existe que dans le cas où la valeur de l'APID vaut l'énumération « UN ».

Pour spécifier que l'élément « TM2 » de l'arbre n'existe que pour l'intervalle de valeurs de l'APID (« DEUX » à « TROIS »), il faudrait passer à la méthode « createAssociationTable », la chaîne de caractère « TELEMETRY/PACKET_HEADER/APID[DEUX..TROIS] ».

7.2.1.8.2. Création et utilisation d'une constante

Le code ci-dessous présente la manière de créer une constante et de l'utiliser dans la définition d'un entier par exemple.

```
// Creation de la constante (max value) "CST_50" du parametre PARAM_ENTIER_1 et ajout
au modele.
XifConstant cst50 = XifConstant.createDefaultXifConstant(mainDataModel);
cst50.setName("CST_50");
cst50.setTypeNameRef(XifConstant.TYPE_NAME_REF_INT);
cst50.setValue("50");
mainDataModelHandler.addConstant(cst50);

// Creation du parametre PARAM_ENTIER_1 et rattachement a la TM1.
XifIntegerType paramEntier1 =
XifIntegerType.createDefaultXifIntegerType(mainDataModel);
// Creation de l'intervalle du type entier.
XifIntegerRange newRange = paramEntier1.getIntegerRange();
newRange.setMinValue(BigInteger.valueOf(-30));
newRange.setMaxConstantNameRef("CST_50");
// Affectation des attributs du type Entier....
```

Les 4 premières lignes du code montrent la manière de créer une constante, la nommer, la typer et lui affecter les valeurs qui la définissent.

Il existe deux possibilités de typage d'une constante XIF, selon l'argument passé à la méthode « setTypeNameRef(String) » :

- Par type « simple » : les constantes (Java) dont le nom débute par « TYPE_NAME_REF_ » de la classe « XifConstant » définissent les noms des types « simples » qui peuvent être utilisés comme argument de cette méthode, comme effectué dans cet exemple.

- Par type XIF : l'argument doit être le nom du type XIF correspondant.

N.B : La méthode « hasBasicType() » permet de déterminer si une constante est de type « simple. Dans le cas contraire, la méthode « getType() » permet d'accéder à la définition du type XIF.

Il est ensuite nécessaire d'utiliser l'instance de la classe « XifDatModelHandler » afin de l'ajouter au modèle via la méthode « addConstant ».

Enfin, l'affectation de cette constante à la valeur du maximum de l'intervalle de l'entier créé ensuite s'effectue par le référencement de cette constante en utilisant son nom. Ainsi, une constante est identifiée par son nom et assure son unicité.

7.2.1.8.3. Affectation d'une fonction de transfert à un paramètre de l'arbre XIF

Le code présenté ci-dessous montre la façon d'affecter une fonction de transfert de type « calibration » à un paramètre de la description XIF.

```
// Creation du parametre PARAM_ENTIER_1.
XifIntegerType paramEntier1 =
XifIntegerType.createDefaultXifIntegerType(mainDataModel);
// Creation de l'intervalle du type entier.
XifIntegerRange newRange = paramEntier1.getIntegerRange();

...
```

```
// Affectation de la référence vers fonction de calibration portée par le type.
XifExtensionUtilities.setExtensionValueFor(paramEntier1, null, "DEFAULTCALIBRATOR");
Node defaultCalibrator =  XifExtensionUtilities.getExtensionFor(paramEntier1,
"DEFAULTCALIBRATOR");
XifExtensionUtilities.createOrSetAttributeNamed(defaultCalibrator, "calibratorRef",
"calibrator1");

// Création du parameter et affectation du type
XifField fieldEntier1 = XifField.createDefaultXifField("PARAM_ENTIER_1",
mainDataModel);
fieldEntier1.setType(paramEntier1);
// Ajout de l'extension "telemetred" sur le parametre pour specifier la nature de
celui-ci.
WriterExtensionUtilities.addDataSourceForParameter(fieldEntier1, "telemetered");
// Ajout du parametre a la liste des parametres.
mainDataModelHandler.addGlobalField(fieldEntier1);
// Creation du field referencant le parametre global pour le rattacher au HEADER.
XifField fieldRefEntier1 = XifField.createReferenceForXifField(fieldEntier1,
mainDataModel);
mainDataModelHandler.addElement(tm1Node, fieldRefEntier1);
```

Ce code ne montre pas comment créer la définition de la fonction de transfert (présenté dans le chapitre 7.2.1.9.1) mais comment référencer sur un paramètre, une fonction de transfert existante dans la liste des fonctions de transfert factorisées.

Les fonctions de transfert qu'elles soient de type « calibration » ou « décalibration » sont rattachées au type du paramètre « XifXXXtype » et non pas au champ représentant le paramètre « XifField ». C'est la raison pour laquelle l'exemple ci-dessus utilise l'instance « paramEntier1 » de type XifIntegerType pour référencer la fonction de transfert, puis ce type est rattaché au champ qui constitue le paramètre.

Les fonctions de transfert étant des éléments appartenant à la définition des extensions du format XIF, les méthodes d'affectation créent une extension qui est ensuite attachée au type « paramEntier1 ». Cette extension est nommée « DEFAULTCALIBRATOR » et porte un attribut « calibratorRef » contenant le nom unique de la fonction de transfert factorisée dans la liste des fonctions de transfert du descriptif XIF.

7.2.1.8.4. Affectation d'une surveillance à un paramètre de l'arbre XIF

Le code présenté ci-dessous montre la façon d'affecter une surveillance « sol » à un paramètre de la description XIF.

```
// Creation du parametre PARAM_ENTIER_2 et rattachement a la TM1.
XifIntegerType paramEntier2 =
XifIntegerType.createDefaultXifIntegerType(mainDataModel);
// Creation de l'intervalle du type entier.
XifIntegerRange newRange2 = paramEntier2. getIntegerRange();

...

XifField fieldEntier2 = XifField.createDefaultXifField("PARAM_ENTIER_2",
mainDataModel);
fieldEntier2.setType(paramEntier2);

// Affectation de la référence vers la surveillance portée par le champ du paramètre.
XifExtensionUtilities.setExtensionValueFor(fieldEntier2, null, "DEFAULTALARM");
Node defaultAlarm =  XifExtensionUtilities.getExtensionFor(fieldEntier2,
"DEFAULTALARM");
XifExtensionUtilities.createOrSetAttributeNamed(defaultAlarm, "alarmRef", "alarm1");
```

```
// Ajout de l'extension "telemetred" sur le parametre pour specifier la nature de celui-ci.
WriterExtensionUtilities.addDataSourceForParameter(fieldEntier2, "telemetered");
// Ajout du parametre a la liste des parametres.
mainDataModelHandler.addGlobalField(fieldEntier2);
// Creation du field referencant le parametre global pour le rattacher au HEADER.
XifField fieldRefEntier2 = XifField.createReferenceForXifField(fieldEntier2, mainDataModel);
mainDataModelHandler.addElement(tm1Node, fieldRefEntier2);
```

Ce code ne montre pas comment créer la définition de la surveillance (présenté dans le chapitre 7.2.1.9.2) mais comment référencer sur un paramètre, une surveillance existante dans la liste des surveillances factorisées.

Les surveillances sont, au contraire des fonctions de transfert, rattachées au champ du paramètre « XifField » et non pas au type « XifXXXType » du paramètre. C'est la raison pour laquelle l'exemple ci-dessus utilise l'instance « fieldEntier2 » de la classe « XifField » pour référencer la surveillance.

Les surveillances étant des éléments appartenant à la définition des extensions du format XIF, les méthodes d'affectation créent une extension qui est ensuite attachée au champ « fieldEntier2 ». Cette extension est nommée « DEFAULTALARM » et porte un attribut « alarmRef » contenant le nom unique de la surveillance factorisée dans la liste des surveillances du descriptif XIF.

7.2.1.8.5. Création de l'élément « TAB »

Le code présenté ci-dessous montre la manière de définir un tableau ou un élément de type «XifArrayType » dans l'arbre et de lui affecter une taille conditionnée par la valeur du paramètre « SIZE_TAB ».

```
// Creation de l'element repete dans le tableau cree ensuite.
XifCharacterType defaultType =
XifCharacterType.createDefaultXifCharacterType(mainDataModel);
// Création de l'intervalle de définition du caractère.
CharacterRangeType range = defaultType.getCharacterRange();
range.setMin(0);
range.setMax(255);

XifField fieldDefault = XifField.createDefaultXifField("DEFAULT", mainDataModel);
fieldDefault.setType(defaultType);

// Ajout de l'extension "telemetred" sur le parametre pour specifier la nature de celui-ci.
WriterExtensionUtilities.addDataSourceForParameter(fieldDefault, "telemetered");

// Ajout du parametre a la liste des parametres.
mainDataModelHandler.addGlobalField(fieldDefault);

// Creation du field referencant le parametre global pour le rattacher au TAB.
XifField fieldRefDefault = XifField.createReferenceForXifField(fieldDefault, mainDataModel);

// Creation du tableau TAB et rattachement a la TM2.
XifArrayType tab = XifArrayType.createDefaultXifArrayType(mainDataModel);
tab.setRepetedField(fieldRefDefault);
XifField fieldTab = XifField.createDefaultXifField("TAB", mainDataModel);
fieldTab.setType(tab);
mainDataModelHandler.addElement(tm2Node, fieldTab);

// Création de la taille conditionnée par la valeur de l'élément "SIZE_TAB"
XifDiscriminantAssociation discDim = new XifDiscriminantAssociation();
```

```
discDim.createAssociationTable(fieldTab, "TELEMETRY/PACKET_DATA/TM2/SIZE_TAB",
fieldTab.getDataModel());

XifStringOrArraySizeType sizeArrayType =
XifStringOrArraySizeType.createDefaultXifStringOrArraySizeType(mainDataModel);
sizeArrayType.setDynamicSizeMode();
sizeArrayType.setConditionAssociation(discDim);
sizeArrayType.setMaxSizeValue(null);
tab.removeDimensionAt(0);
tab.addDimension(sizeArrayType);
```

La première partie de l'exemple ci-dessus montre la création d'un paramètre de type « XifCharacterType » utilisé pour constituer l'élément répété dans le tableau construit par la suite.

La seconde partie montre la manière de créer un tableau et de lui affecter l'élément de type « XifCharacterType » créé précédemment.

Enfin, la dernière partie montre la création de la taille dynamique du tableau conditionnée par l'élément dont le chemin dans l'arbre XIF est « TELEMETRY/PACKET_DATA/TM2/SIZE_TAB ». On voit ici que la manière de définir une taille dynamique de tableau est identique à la création d'une condition d'existence sur un élément à la différence près que l'on ne donne que le chemin de l'élément discriminant et non pas le couple chemin et valeur de l'élément discriminant. Cela revient donc à dire que la taille du tableau est discriminée par n'importe quelle valeur de l'élément discriminant.

Il existe trois types XIF dont la taille peut être définie de cette manière : « XifArrayType », « XifStringType » et « XifGenericType ».

7.2.1.9. Création des extensions globales

Un certain nombre d'extensions dédiées au « Monitoring And Control » ont été conçues de manière à être factorisées. On entend par « factorisées », le fait d'être créées de manière globale sans être affectées à un paramètre quelconque, pour ensuite être référencées au niveau des paramètres ou autres éléments de l'arbre de description XIF et ce de manière unique ou multiple.

Ces extensions globales sont stockées dans le fichier XIF au niveau du nœud « generalInformation » qui centralise les informations générales du XIF comme son nom l'indique.

Les avantages de ce procédé sont :

- de pouvoir définir un ensemble d'extensions telles que des fonctions de transferts ou des surveillances, par exemple, et ensuite, de les utiliser à plusieurs reprises sur différents paramètres qui auraient besoin de la même définition de fonction de transferts ou de surveillance.

- de réduire la taille des fichiers XIF en ne définissant qu'une fois l'extension puis en la référençant aux différents endroits nécessaires sans recopier la définition.

- de centraliser les définitions,

- de minimiser les impacts lorsqu'une extension est modifiée et donc d'améliorer les performances.

Les principales extensions du mode « Monitoring And Control » que sont les fonctions de calibration, de décalibration et les surveillances fonctionnent sur ce principe.

L'ensemble des extensions est détaillé dans le document DR2 .

Le détail de l'utilisation des principales extensions du mode « Monitoring And Control » est expliqué dans le document DR3.

Dans l'exemple que nous suivons ici, nous allons créer deux extensions représentatives du mode « Monitoring And Control » de manière globale, et qui seront par la suite référencées dans l'arbre de description du XIF.

7.2.1.9.1. Création d'une fonction de transfert

Il est possible de créer différents types de fonction de calibration, l'exemple illustré ici s'appuie sur une fonction de calibration de type polynomiale :

```
// Creation des valeurs du polynome
Hashtable<String, String> aList = new Hashtable<String, String>();
aList.put("0.0", "0");
aList.put("5.0", "1");
// Appel de la methode de creation de la fonction de calibration
WriterExtensionUtilities.addGlobalPolynomialCalibrator(mainDataModel.getGeneralInform
ation(), "calibrator1", null, aList);
```

Dans l'exemple ci-dessus, on affecte tout d'abord les valeurs du polynôme de la fonction de calibration à une collection afin de stocker ces valeurs pour ensuite les affecter à la définition de l'extension.

Cette collection est ensuite transmise à la méthode permettant de créer l'extension correspondant à la fonction de calibration.

L'exemple suivant montre la déclaration de la méthode précédemment citée et ainsi montre une manière de créer les extensions de type « fonction de calibration » dans le XIF.

```
public static void addGlobalPolynomialCalibrator(XifGeneralInformation
aGeneralInformation, String aName, String aDesc, Hashtable<String, String> aList)
throws Exception {
  // récupération de la liste des calibrateurs
  Node globCalExt =
XifExtensionUtilities.getNeverNullExtensionFor(aGeneralInformation,
"CALIBRATORLIST");

  // création du nouveau calibrateur
  Node calNode = XifExtensionUtilities.createAndAddChildNamed(globCalExt,
"CALIBRATOR");

  // ajout du nom du calibrateur
  XifExtensionUtilities.createOrSetAttributeNamed(calNode, "name", aName);

  // ajout de la description
  XifExtensionUtilities.createOrSetAttributeNamed(calNode, "shortDescription",
aDesc);

  // creation du polynome
  Node polynomialNode = XifExtensionUtilities.createAndAddChildNamed(calNode,
"POLYNOMIAL");
  Set<String> keys = aList.keySet();
  for (Iterator<String> iter = keys.iterator(); iter.hasNext();) {
     String pt = iter.next();
     // ajout des coefficients et facteurs du polynome
     Node termNode = XifExtensionUtilities.createAndAddChildNamed(polynomialNode,
"TERM");
```

```
     XifExtensionUtilities.createOrSetAttributeNamed(termNode, "coefficient", pt);
     XifExtensionUtilities.createOrSetAttributeNamed(termNode, "exponent",
aList.get(pt));
  }
}
```

Les extensions « globales » étant attachées au nœud « generalInformation » du descriptif XIF, l'objet « XifGeneralInformation » est passé en paramètre de la méthode.

De manière générale, la méthode est construite en s'assurant que des erreurs ne peuvent pas être produites au cours de la construction de l'extension. On s'assure donc, lorsque l'on récupère l'extension sur l'objet « generalInformation », que celle-ci n'est pas initialisée (valeur « null »). Si c'est le cas, cela signifie qu'elle n'a pas encore été créée, elle est donc créée automatiquement ; de même pour le nœud « CALIBRATORLIST » contenant la liste des « CALIBRATOR » correspondant à chaque fonction de calibration factorisée.

La création de la description de la fonction de calibration s'appuie ensuite sur la définition XSD des extensions XIF dans laquelle on construit un sous arbre XML formé de nœud contenant des valeurs, des attributs et des nœuds fils. On s'appuie sur des méthodes statiques fournies par l'API OASIS dans la classe « XifExtensionUtilities » du paquetage « modeller.common » qui fournit un bon nombre de méthodes de manipulation génériques des extensions.

Il est donc possible de créer tout type d'extension sur la base de cet exemple en reprenant la définition XSD de l'extension que l'on souhaite créer ou manipuler et en s'appuyant sur le détail de leur définition à partir des documents DR2 et DR3.

De la même manière, il est donc possible de créer une fonction de surveillance comme le paragraphe suivant le détaille.

7.2.1.9.2. Création d'une fonction de surveillance

De la même manière que l'on créé une fonction de calibration dans le paragraphe précédent, le procédé pour créer une surveillance est sensiblement identique :

```
// Appel de la methode de creation de la surveillance
WriterExtensionUtilities.addGlobalGroundAlarm(mainDataModel.getGeneralInformation(),
"alarm1", null, alarmLevels.warning, alarmType.numeric, "10", "20", "seconds");
```

La déclaration de la méthode est la suivante :

```
public static void addGlobalGroundAlarm(XifGeneralInformation aGeneralInformation,
String alarmName, String alarmDesc, alarmLevels alarmLevel,      alarmType alarmType,
String minInclusive, String maxInclusive, String timeUnits) throws Exception {

  Node globAlarmListExt =
XifExtensionUtilities.getNeverNullExtensionFor(aGeneralInformation, "ALARMLIST");

  // creates a new alarm
  Node alarmNode = XifExtensionUtilities.createAndAddChildNamed(globAlarmListExt,
"ALARM");
  // creates name attribute
  XifExtensionUtilities.createOrSetAttributeNamed(alarmNode, "name", alarmName);
  // creates type attribute
  XifExtensionUtilities.createOrSetAttributeNamed(alarmNode, "type",
alarmType.toString());

  // creates a new definition
```

```
  Node definitionNode = XifExtensionUtilities.createAndAddChildNamed(alarmNode,
"DEFINITION");

  switch (alarmType) {
  case numeric : {
    // creates a new numericAlarm
    Node numericAlarmNode =
XifExtensionUtilities.createAndAddChildNamed(definitionNode, "NUMERICALARM");

    // creates a new staticAlarmRanges
    Node staticAlarmRangesNode =
XifExtensionUtilities.createAndAddChildNamed(numericAlarmNode, "STATICALARMRANGES");

    // creates a new staticAlarmRange
    Node staticAlarmRangeNode =
XifExtensionUtilities.createAndAddChildNamed(staticAlarmRangesNode,
"STATICALARMRANGE");
    // creates alarmLevel attribute
    XifExtensionUtilities.createOrSetAttributeNamed(staticAlarmRangeNode,
"alarmLevel", alarmLevel.toString());

    // creates a new verifyRange
    Node verifyRangeNode =
XifExtensionUtilities.createAndAddChildNamed(staticAlarmRangeNode, "VERIFYRANGE");
    // creates verifyRange attributes
    XifExtensionUtilities.createOrSetAttributeNamed(verifyRangeNode, "mininclusive",
minInclusive);
    XifExtensionUtilities.createOrSetAttributeNamed(verifyRangeNode, "maxinclusive",
maxInclusive);

    // creates a new timeunits
    XifExtensionUtilities.createAndAddChildNamed(verifyRangeNode, "TIMEUNITS",
timeUnits);
  }
  case enumeration : {
    // creates a new enumeration Alarm
    // ....

  }
  case string : {
    // creates a new string Alarm
    // ....
  }
  }
}
```

Le procédé est donc sensiblement identique à celui des fonctions de calibration dans le sens où l'on s'assure de l'existence du nœud principal des extensions sur l'objet « XifGeneralInformation » ainsi que de l'existence de la liste des alarmes « ALARMLIST », puis l'on créé la définition de l'alarme en s'appuyant sur la définition XSD de celle-ci et toujours avec les méthodes fournies par l'API OASIS.

7.2.1.10. Validation des extensions d'un objet XIF

Les extensions affectées à un objet XIF peuvent être validées de la manière suivante :

```
ExtensionErrorHandler handler = new ExtensionErrorHandler();
XifGeneralInformation generalInfos = model.getGeneralInformation();
```

```
boolean checkResult = XifExtensionUtilities.check(generalInfos.getExtension(),
handler, null);
if (!checkResult) {
	for (String errorMsg : handler.getErrors()) {
		// TODO traiter chaque erreur enregistrée
	}
}
```

Cet exemple effectue une validation des extensions affectées à l'élément « XifGeneralInformation » du modèle. La validation peut s'effectuer sur toute extension affectée à un objet XIF et accessible par la méthode « getExtension ».

7.2.1.11. Sauvegarde du descriptif XIF

La dernière étape à réaliser une fois la description complète modélisée en mémoire, est de sauvegarder le modèle dans le format XIF. Le code ci-dessous permet de réaliser cette action :

```
// Sauvegarde du XIF
xifResource.save();
```

La méthode « save() » de l'instance de la classe « XifResource » permet de sauvegarder le fichier XIF avec le nom et dans le répertoire, spécifier au modèle de données principal.

Il est également possible de spécifier le chemin et le nom du fichier XIF directement à la sauvegarde. Le chemin et le nom du fichier stocké dans le modèle de données sont alors écrasés :

```
// Sauvegarde du fichier XIF avec un chemin spécifique
File xifFile = new File("D:\\MON_PROJET\\DATA_MODEL\\monXif.xif");
xifResource.saveAs(xifFile);
```

7.2.2. Référencement d'un ou plusieurs XIF à partir d'un XIF « maître »

7.2.2.1. Référencer un fichier XIF

La classe « XifResource » fournit la méthode « attachDataModel(String aFilePath) » afin de charger un fichier XIF dont le chemin est représenté par la chaîne « aFilePath ».

Cette méthode vérifie que le fichier XIF peut être chargé, c'est-à-dire que le fichier XIF existe bien et qu'il contient un « targetNameSpace ».

Lorsqu'un sous modèle est chargé dans la ressource « XifResource », l'ensemble des constantes, types ou encore champs globaux sont accessibles depuis les méthodes offertes par la classe « XifResource ».

Par exemple :

Cette méthode (de la classe « XifResource ») retourne l'ensemble des types internes du modèle principal ainsi que des modèles directement référencés :

```
public List<XifObject> getInternalTypeList();
```

Cette méthode (de la classe « XifResource ») retourne l'ensemble des types internes du modèle principal ainsi que des modèles référencés et également les types des modèles référencés de troisième niveau et plus (cas où un XIF référencé référence lui-même un autre XIF et ainsi de suite …).

```
public List<XifObject> getAllInternalTypeList();
```

7.2.2.2. Retirer la référence sur un fichier XIF

La classe « XifResource » fournit la méthode « detachDataModel(XifDataModel aDataModel) » afin de retirer la référence d'un fichier XIF sur le modèle de données principal porté par la ressource.

Cette méthode vérifie que le fichier XIF peut être retiré, c'est-à-dire qu'aucune constante, qu'aucun type ou aucun champ global de cet XIF n'est utilisé dans l'arbre du modèle principal porté par la ressource.

7.3. COPIE D'UNE PARTIE DE FICHIER XIF

Ce chapitre montre la démarche à adopter pour copier une partie d'un arbre XIF et coller cette copie à un endroit spécifique de l'arbre.

L'exemple de ce chapitre s'appuie sur le fichier XIF présenté en Annexe B de ce document. Dans cet exemple, on copie la branche de l'arbre nommé « TM1 », on créé un champ « TM3 » de type « XifUndefined » attaché au champ « PACKET_DATA » puis on colle l'élément copié sur ce nouveau champ « TM3 ».

Les sources de la classe JAVA sur laquelle s'appuie l'exemple proposé ci-après se trouve en Annexe D de ce document.

7.3.1. Copie d'un champ de l'arbre XIF

Le code suivant montre la manière de copier un élément de l'arbre XIF en s'appuyant sur le nœud « XifNode ».

```
protected XifCopiedType copyType(XifNode aNode) throws Exception {
  // Récupération du modèle de données.
  XifDataModel dataModel = aNode.getDataModel();
  // Récupération du champ contenu par le noeud.
  XifObject obj = aNode.getUserObject();
  if (obj.isField()) {
    // Récupération du type du champ.
    XifObject type = obj.getType();
    // On vérifie que le type n'esy pas un type undéfini
    // sinon il n'y a pas d'intérêt à effectuer la copie.
    if (!type.isUndefinedType()) {
      // Chaque type offre une méthode de copie de lui-même.
      XifObject copiedType = type.copy(dataModel, true);
      // On effectue la copie des discriminant
      // si il en existe dans la structure copiée.
      ArrayList<XifCopiedConditionForDiscriminant> discriminantsToKeepList = new
ArrayList<XifCopiedConditionForDiscriminant>();
      Utilities.completeCopiedType(type, copiedType, discriminantsToKeepList);
      // On retourne la nouvelle instance de l'élément copié.
      return new XifCopiedType(copiedType, dataModel, discriminantsToKeepList,
(XifField)obj);
    }
    else {
      throw new Exception("Le type est un type undefini ....");
    }
  }
  else {
    throw new Exception("La copie n'est possible que sur un XifField");
  }
}
```

Chaque type « XifXXXType » de l'API OASIS contient une méthode « copy() » permettant d'effectuer une copie de lui-même et ainsi utiliser cette copie dans un autre contexte.

La méthode « completeCopiedType() » de la classe « Utilities » de l'API OASIS permet de parcourir récursivement

le type si celui-ci est une structure et de récupérer les conditions d'existence qui peuvent y être définies pour pouvoir les embarquer dans l'élément copié.

La classe « XifCopiedType » permet de transporter le type copié et fournit les méthodes nécessaires à la recopie de celui-ci.

7.3.2. Re-copie de l'élément

Le code suivant montre la manière de coller un élément copié dans le même XIF.

Pour effectuer une re-copie d'un élément, il est nécessaire que l'élément destination soit un champ dont le type est undéfini. C'est une règle qui permet de se prémunir d'un écrasement de type défini involontaire.

```
// Copie de l'élément "TM1", création du champ "TM3" undéfini
// puis recopie de l'élément dans le nouveau champ.
if (packetDataNode != null) {
  XifNode tm1Node = packetDataNode.getChildNodeNamed("TM1");
  if (tm1Node != null) {
    tm1Copy = copyType(tm1Node);

    // Creation d'un element TM3 sur le noeud "PACKET_DATA"
    XifUndefinedType tm3 = XifUndefinedType.createDefaultUndefined(mainDataModel);
    XifField fieldTm3 = XifField.createDefaultXifField("TM3", mainDataModel);
    fieldTm3.setType(tm3);
    mainDataModelHandler.addElement(packetDataNode, fieldTm3);
    XifNode tm3Node = packetDataNode.getChildNodeNamed("TM3");

    // Collage de l'élément copié.
    XifObject typeToPaste = tm1Copy.getCopiedType();
    mainDataModelHandler.modifyElement(tm3Node, typeToPaste);

    // Copie et mise à jour des conditions d'existence.
    if (tm1Copy.getDiscriminantAssociationList() != null) {
      ArrayList<XifCopiedConditionForDiscriminant>
listOfDiscriminantAssociationToKeep = tm1Copy.getDiscriminantAssociationList();
      Utilities.completeConditionDiscriminantInTable(tm3Node,
listOfDiscriminantAssociationToKeep);
      Utilities.createConditionDiscriminantOfCopiedType(typeToPaste,
listOfDiscriminantAssociationToKeep, mainDataModel);
    }
  }
}
```

La première partie de ce code montre la manière de créer un nouvel élément dans l'arbre nommé « TM3 » et de lui affecter un type undéfini « XifUndefinedType ».

La seconde partie montre la récupération de l'objet copié puis l'affectation de ce type copié au nouvel élément « TM3 » de l'arbre.

7.3.3. Re-copie de l'élément dans un autre descriptif XIF

Le code suivant montre la manière de coller un élément copié d'un descriptif XIF vers un autre descriptif XIF.

Pour effectuer une telle re-copie, il est nécessaire que l'élément destination soit un champ dont le type est undéfini. C'est une règle qui permet de se prémunir d'un écrasement de type défini involontaire.

```
// Copie de l'élément source
XifCopiedType copiedType = copyType(copiedNode);

// Récupération de l'élément copié.
XifObject typeToPaste = copiedType.getCopiedType();

// On vérifie que le type de l'élément copié n'existe pas dans le XIF cible
// si celui-ci est nommé.
ArrayList<XifObject> typeNamedInCopy = typeToPaste.getTypeNamedIn();
for (int i = 0; i < typeNamedInCopy.size(); i++) {
  XifObject aType = typeNamedInCopy.get(i);
  if (targetXifResource.getInternalTypeNamed(aType.getName()) != null) {
    throw new Exception("Le type copié existe déjà dans le modèle cible ....");
  }
}
// Le type doit être cloné pour être transporté.
typeToPaste = XifTypeManager.clone(typeToPaste, targetMainDataModel);
typeNamedInCopy = typeToPaste.getTypeNamedIn();
// Ajout du type dans la liste des types du descriptif cible.
for (int j = 0; j < typeNamedInCopy.size(); j++) {
  targetMainDataModel.getDataModelHandler().addInternalType(typeNamedInCopy.get(j));
}

// Rattachement de l'élément à la cible.
targetMainDataModel.getDataModelHandler().modifyElement(pastNode, typeToPaste);
if (copiedType.getDiscriminantAssociationList() != null) {
  ArrayList<XifCopiedConditionForDiscriminant> listOfDiscriminantAssociationToKeep =
copiedType.getDiscriminantAssociationList();
  Utilities.completeConditionDiscriminantInTable(pastNode,
listOfDiscriminantAssociationToKeep);
  Utilities.createConditionDiscriminantOfCopiedType(typeToPaste,
listOfDiscriminantAssociationToKeep, targetMainDataModel);
}
// Sauvegarde du modèle modifié dans un nouveau XIF.
targetXifResource.save();
```

Ce code montre qu'il est nécessaire de penser à copier le type de l'élément vers le descriptif cible car celui-ci peut être une référence. Dans ce cas, il est nécessaire de recopier le type dans le descriptif cible pour que la référence du type collé ne pointe pas vers un type inexistant dans le XIF cible. On vérifie au préalable qu'un type ne porte pas déjà ce même nom dans le XIF cible.

ANNEXE A : SOURCES DES CLASSES JAVA DE L'EXEMPLE DE LECTURE

```
package fr.cs.best.modeller.reader;

import java.math.BigInteger;

import modeller.ModellerServicesInterface;
import modeller.common.CharacterCheck;
import modeller.common.XifPreProcessing;
import modeller.core.dataManager.xifManager.XifDataModel;
import modeller.core.dataManager.xifManager.XifDiscriminantAssociation;
import modeller.core.dataManager.xifManager.XifResource;
import modeller.core.dataManager.xifManager.api.XifArrayType;
import modeller.core.dataManager.xifManager.api.XifCharacterType;
import modeller.core.dataManager.xifManager.api.XifDocumentation;
import modeller.core.dataManager.xifManager.api.XifDocumentationParameter;
import modeller.core.dataManager.xifManager.api.XifEnumeratedType;
import modeller.core.dataManager.xifManager.api.XifExtension;
import modeller.core.dataManager.xifManager.api.XifField;
import modeller.core.dataManager.xifManager.api.XifGeneralInformation;
import modeller.core.dataManager.xifManager.api.XifGenericType;
import modeller.core.dataManager.xifManager.api.XifIntegerType;
import modeller.core.dataManager.xifManager.api.XifNode;
import modeller.core.dataManager.xifManager.api.XifObject;
import modeller.core.dataManager.xifManager.api.XifRealType;
import modeller.core.dataManager.xifManager.api.XifRecordType;
import modeller.core.dataManager.xifManager.api.XifStringOrArraySizeType;
import modeller.core.dataManager.xifManager.api.XifStringType;
import modeller.core.dataManager.xifManager.api.XifUndefinedType;

import org.w3c.dom.Element;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.Node;

/**
 * Exemple de lecture de fichier XIF en s'appuyant sur l'API OASIS.
 * Le résultat de l'extraction des éléments du XIF sont renvoyé sur la sortie
standard.
 */
public class XifReader {

	private XifResource xifResource;

	private static final String FICHIER_XIF = "xif/DOR_HKTM.xif";

	public XifReader() {
	}

	/**
	 * Main de la classe.
	 * @param args les arguments passés en ligne de commande.
	 */
	public static void main(String[] args) {
```

```
            try {

                  preProcessXifVersion();

                  XifResource resource =
ModellerServicesInterface.loadXmlIf(FICHIER_XIF);

                  postProcessXifVersion(resource);

                  // parse le ficheir XIF
                  new XifReader().parseXif(resource);
            }
            catch (Exception e) {
                  System.out.println("[ERROR] Unable to load the current XIF file.
\n" + e.getMessage());
            }
      }

      /**
       * Permet d'extraire une information du XIF sans le cahrger via l'API SAX.
       */
      private static void preProcessXifVersion() {
            XifPreProcessing xifProcessing = new XifPreProcessing("modellerVersion",
FICHIER_XIF);
            xifProcessing.startParsing();
            String modellerVersion = xifProcessing.getReadValue();

            XifPreProcessing xifProcessing2 = new
XifPreProcessing("xifSchemaVersion", FICHIER_XIF);
            xifProcessing2.startParsing();
            String xifSchemaVersion = xifProcessing2.getReadValue();

            System.out.println(modellerVersion + " - " + xifSchemaVersion);
      }

      /**
       * Permet d'extraire la même information que le preProcess mais via l'API
OASIS.
       * @param resource la resource.
       */
      private static void postProcessXifVersion(XifResource resource) {
            XifDataModel datamodel = resource.getXifDataModel();
            XifGeneralInformation generalInformation = (XifGeneralInformation)
datamodel.getInternalForm().getGeneralInformation();

            // vérification de la version de l'api oasis
            String fileVersionXif =
generalInformation.getVersions().getModellerVersion();

            // vérification de la version du schéma xif
            String fileVersionSchemaXif =
generalInformation.getVersions().getXifSchemaVersion();

            System.out.println(fileVersionXif + " - " + fileVersionSchemaXif);
      }

      /**
       * Parse le ficheir XIF.
       * @param xifResource la resource XIF.
       */
```

```
	public void parseXif(XifResource aResource) {
		xifResource = aResource;
		System.out.println("--- STARTING XIF PARSING ---");
		System.out.println("-> Model name = " +
xifResource.getResourceFilename());
		parse();
	}

	/**
	 * Boucle sur les noeuds du XIF.
	 */
	private void parse() {
		XifNode root = null;
		int nbRoots = xifResource.getXifDataModel().getRootCount();
		System.out.println("-> Numbers of roots = " + nbRoots);
		System.out.println();
		for (int i = 0 ; i < nbRoots ; i++) {
			root = xifResource.getXifDataModel().getRootAt(i);
			System.out.println("processing root named = " + root.getName());
			parseNode(root);
		}
	}

	/**
	 * parse le noeud courant.
	 * @param aChild le noeud à parser.
	 */
	private void parseNode(XifNode aChild) {

		XifObject field = aChild.getUserObject();
		if (field.isRoot()) {
			System.out.println("--- Processing root named = " + field.getName()
+ "---");
		} else {
			System.out.println("--- Processing field named = " +
field.getName() + "---");
		}

		// Si une condition d'existence existe, on la traite.
		if (aChild.isOptionalNode()) {
			parseOptionality(aChild);
		}

		// recupère le type.
		XifObject type = field.getType();
		// parse le type.
		if (type.isArrayType()) {
			parseArray(aChild, (XifArrayType)type);
		}
		else if (type.isCharacterType()) {
			parseCharacter(aChild, (XifCharacterType)type);
		}
		else if (type.isEnumeratedType()) {
			parseEnumerated(aChild, (XifEnumeratedType)type);
		}
		else if (type.isIntegerType()) {
			parseInteger(aChild, (XifIntegerType)type);
		}
		else if (type.isListType()) {
```

```
                parseNode(aChild.getChildNodes().get(0));
            }
            else if (type.isRealType()) {
                parseReal(aChild, (XifRealType)type);
            }
            else if (type.isRecordType()) {
                parseRecord(aChild, (XifRecordType)type);
            }
            else if (type.isStringType()) {
                parseString(aChild, (XifStringType)type);
            }
            else if (type.isUndefinedType()) {
                parseUndefined(aChild, (XifUndefinedType)type);
            }
            else if (type.isGenericType()) {
                parseGeneric(aChild, (XifGenericType)type);
            }
            parseDocumentation(aChild, type);
            parseExtension(aChild, type);
            System.out.println("--- End processing field named = " + field.getName()
+ " ---");
        }

        private void parseUndefined(XifNode aChild, XifUndefinedType type) {
            System.out.println("[UNDEFINED][UNDEFINED]");
        }

        private void parseGeneric(XifNode aChild, XifGenericType type) {
            System.out.println("[GENERIC]");
            if (type.getSize().isDynamicSizeMode()) {
                parseDynamicSize(type, aChild);
            }
            else {
                System.out.println("Size is = " + type.getBitsSize());
            }
            System.out.println("[GENERIC]");
        }

        private void parseString(XifNode aChild, XifStringType type) {
            System.out.println("[STRING]");
            if (type.getStringSize().isDynamicSizeMode()) {
                parseDynamicSize(type, aChild);
            }
            else {
                System.out.println("Size is = " + type.getBitsSize());
            }
            System.out.println("[STRING]");
        }

        private void parseArray(XifNode aChild, XifArrayType type) {
            System.out.println("[ARRAY] Repeted element is = " +
aChild.getChildNodes().get(0).getName());
            parseDynamicSize(type, aChild);
            parseNode(aChild.getChildNodes().get(0));
            System.out.println("[ARRAY]");
        }

        private void parseEnumerated(XifNode aChild, XifEnumeratedType anEnum) {
            StringBuilder sb = new StringBuilder();
```

```
            int nbVal = anEnum.getNumberOfValues();
            // range mode
            sb.append("[ENUMERATION] number of values = ");
            sb.append(nbVal);
            sb.append(" : ");

            for (int i = 0; i < nbVal ; i++) {
                  sb.append(anEnum.getValueAt(i));
                  if (i+1 < nbVal) {
                        sb.append(", ");
                  }
            }

            sb.append(" [ENUMERATION]");
            System.out.println(sb.toString());
      }

      private void parseCharacter(XifNode aChild, XifCharacterType aCharacter) {
            StringBuilder sb = new StringBuilder();

            sb.append("[CHARACTER] Min = ");

            int begin = aCharacter.getBeginCharacter();
            int end = aCharacter.getEndCharacter();
            // min
            if (CharacterCheck.isPrintable((char)begin)) {
                  sb.append("'" + (char)begin + "'");
            }
            else {
                  String minStr = Integer.toString(begin);
                  if (minStr.length() == 1) {
                        minStr = "00" + minStr;
                  }
                  if (minStr.length() == 2) {
                        minStr = "0" + minStr;
                  }
                  sb.append("\\" + minStr);
            }
            sb.append(", Max = ");
            // max
            if (CharacterCheck.isPrintable((char)end)) {
                  sb.append("'" + (char)end + "'");
            }
            else {
                  String maxStr = Integer.toString(end);
                  if (maxStr.length() == 1) {
                        maxStr = "00" + maxStr;
                  }
                  if (maxStr.length() == 2) {
                        maxStr = "0" + maxStr;
                  }
                  sb.append("\\" + maxStr);
            }
            sb.append(" [CHARACTER]");
            System.out.println(sb.toString());
      }

      /**
       * Parse le type eniter.
```

```
         * @param aNode l'élément portant le type à traiter.
         * @param anInteger le type à traiter.
         */
        private void parseInteger(XifNode aNode, XifIntegerType anInteger) {
                StringBuilder sb = new StringBuilder();

                sb.append("[INTEGER] Range mode = ");
                if (anInteger.isBeginEndRange()) {
                        sb.append("BEGIN_END, ");
                }
                else if (anInteger.isFullRange()){
                        sb.append("FULL_RANGE, ");
                }
                else {
                        sb.append("BEGIN_SIZE, ");
                }
                BigInteger beginValue = anInteger.getRange().getBegin();
                sb.append("begin : " + beginValue);

                BigInteger endValue = anInteger.getRange().getEnd();
                sb.append(", end : " + endValue);

                BigInteger sizeValue = anInteger.getRange().getSize();
                sb.append(", size : " + sizeValue);

                long bitsSize = anInteger.getBitsSize();
                sb.append(", size in bits is = " + bitsSize);

                sb.append(", encoding is = ");
                if (anInteger.isAscii()) {
                        sb.append("ASCII. [INTEGER]");
                }
                else {
                        sb.append("BINARY. [INTEGER]");
                }
                System.out.println(sb.toString());
        }

        private void parseReal(XifNode aNode, XifRealType aReal) {
                StringBuilder sb = new StringBuilder();

                sb.append("[REAL] Range mode = ");
                if (aReal.isBeginEndRange()) {
                        sb.append("BEGIN_END, ");
                }
                else if (aReal.isBeginEndPrecisionRange()){
                        sb.append("BEGIN_END_PRECISION, ");
                }
                else {
                        sb.append("FULL_RANGE, ");
                }

                sb.append("size is = ");
                sb.append(aReal.getBitsSize());

                sb.append(" bits, encoding is = ");
                if (aReal.isAscii()) {
                        sb.append(", encoding is = ASCII. [REAL]");
                }
                else {
```

```
                        sb.append(", encoding is = BINARY. [REAL]");
                }
                System.out.println(sb.toString());
        }

        /**
         * Parse le type record.
         * @param aNode l'élément portant le type à traiter.
         * @param aRecord le type à traiter.
         */
        private void parseRecord(XifNode aNode, XifRecordType aRecord) {
                StringBuilder sb = new StringBuilder();
                int nbChildren = aRecord.getNumberOfFields();

                sb.append("[RECORD] Has = ");
                sb.append(nbChildren);
                sb.append(" children, children list = ");
                System.out.println(sb.toString());
                for (int i = 0 ; i < nbChildren ; i++) {
                        XifNode child = aNode.getChildNodeAt(i);
                        parseNode(child);
                }
                sb.setLength(0);
                sb.append("[RECORD]");
                System.out.println(sb.toString());
        }

        /**
         * Parse la condition d'existence du noeud.
         * @param aNode le noeud portant la condition.
         */
        private void parseOptionality(XifNode aNode) {
                XifField field = (XifField)aNode.getUserObject();
                XifDiscriminantAssociation discAssos = field.getConditionAssociation();
                if (discAssos != null) {
                        System.out.println("[OPTIONALITY] on node " + aNode + " is " +
discAssos.getDiscriminationValue(aNode));
                }
        }

        /**
         * Parse la taille dynamique d'un élément de type String ou Array.
         * @param aType le type à traiter (string ou array).
         * @param aNode le noeud portant la condition.
         */
        private void parseDynamicSize(XifObject aType, XifNode aNode) {
                if (aType.isStringType()) {
                        XifStringOrArraySizeType strSize =
((XifStringType)aType).getStringSize();
                        XifDiscriminantAssociation discAssos =
strSize.getConditionAssociation();
                        if (discAssos != null) {
                                System.out.println("[DYNAMICSIZE] on node " + aNode + " is "
+ discAssos.getDiscriminationValue(aNode));
                        }
                }
                else if (aType.isArrayType()) {
                        XifArrayType array = (XifArrayType)aType;
                        for (int i = 0 ; i < array.getArrayDimensionRange().size() ; i++) {
```

```
                        XifStringOrArraySizeType dimension =
(XifStringOrArraySizeType)array.getArrayDimensionRange().get(i);
                        XifDiscriminantAssociation discAssos =
dimension.getConditionAssociation();
                        if (dimension.isDynamicSizeMode()){ // BEST/FT/0155 : Perte
formule taille variable
                              System.out.println("[DYNAMICSIZE] on node " + aNode + "
is " + discAssos.getDiscriminationValue(aNode));
                        }
                  }
            }
      }

      private void parseDocumentation(XifNode aNode, XifObject xifObject) {
            XifDocumentation xifDoc = (XifDocumentation)
xifObject.getDocumentation();
            if (xifDoc != null) {
                  System.out.println("[DOCUMENTATION] for type named " +
aNode.getName());
                  for (int i = 0; i < xifDoc.getNumberOfDocumentationParameters();
i++) {
                        XifDocumentationParameter param =
xifDoc.getDocumentationParameterAt(i);
                        String semanticAttributeName = param.getName();
                        String semanticAttributeValue = param.getValue();
                        String semanticAttributeContext = param.getContext();
                        System.out.println("[DOCUMENTATION PARAMETER] : " +
semanticAttributeName);
                        System.out.println("                    Value : " +
semanticAttributeValue);
                        System.out.println("                  Context : " +
semanticAttributeContext);
                  }
            }
      }

      private void parseExtension(XifNode aNode, XifObject xifObject) {
            XifExtension xifExtension = (XifExtension) xifObject.getExtension();
            if ((xifExtension != null) && (xifExtension.getNumberOfParameters() > 0))
{
                  System.out.println("[EXTENSION] for type named " +
aNode.getName());
                  if(xifExtension.getNumberOfParameters() > 0) {
                        Node anExtensionNode = xifExtension.getParameterAt(0);
                        if (anExtensionNode instanceof Element) {
                              parseExtensionNode(anExtensionNode, 0);
                        }
                  }
            }
      }

      private void parseExtensionNode(Node aNode, int space) {
            String indent = "";
            for(int i = 0; i < space; i++) {
                  indent += " ";
            }
            System.out.println(indent + "[EXTENSION NODE] : " + aNode.getNodeName());
            System.out.println(indent + "    Text content : " +
aNode.getNodeValue());
            NamedNodeMap map = aNode.getAttributes();
```

```
            if ((map != null) &&(map.getLength() > 0)) {
                  for (int i = 0; i < map.getLength(); i++) {
                        Node attribute = map.item(i);
                        System.out.println(indent + "  Attribute name : " +
attribute.getNodeName());
                        System.out.println(indent + "  Att. content    : " +
attribute.getNodeValue());
                  }
            }
            for (int i = 0; i < aNode.getChildNodes().getLength(); i++) {
                  Node child = aNode.getChildNodes().item(i);
                  if (child instanceof Element) {
                        parseExtensionNode(child, space + 3);
                  }
            }
      }
}
```

ANNEXE B : FICHIER XIF RESULTAT DE L'ECRITURE


```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<internalFormat>
   <generalInformation>
      <versions>
         <xifSchemaVersion>3.0</xifSchemaVersion>
         <extensionSchemaVersion>2.0</extensionSchemaVersion>
         <modellerVersion>8.2</modellerVersion>
         <internalFormatVersion>1.0</internalFormatVersion>
         <pluginsVersion>
            <plugin version="1.0" pluginId="M_C"/>
         </pluginsVersion>
      </versions>
      <xifModelId>M_C</xifModelId>
      <physicalInformation wordSize="16" arrayArranging="ROW" norm="FCSTC000" sendingMachine="PC()"/>
      <extension>
         <EXTENSIONS>
            <CALIBRATORLIST>
               <CALIBRATOR name="calibrator1">
                  <POLYNOMIAL>
                     <TERM coefficient="0.0" exponent="0"/>
                     <TERM coefficient="5.0" exponent="1"/>
                  </POLYNOMIAL>
               </CALIBRATOR>
            </CALIBRATORLIST>
            <ALARMLIST>
               <ALARM name="alarm1" type="numeric">
                  <DEFINITION>
                     <NUMERICALARM>
                        <STATICALARMRANGES>
                           <STATICALARMRANGE alarmLevel="warning">
                              <VERIFYRANGE maxinclusive="20" mininclusive="10"/>
                           </STATICALARMRANGE>
                           <TIMEUNITS>seconds</TIMEUNITS>
                        </STATICALARMRANGES>
                     </NUMERICALARM>
                  </DEFINITION>
               </ALARM>
            </ALARMLIST>
         </EXTENSIONS>
      </extension>
   </generalInformation>
   <constantList>
      <constant value="50" typeNameRef="INTEGER" name="CST_50"/>
   </constantList>
   <typeList>
      <enumerationType size="1" computationMode="AT_THE_BEST" binaryRepresentationMode="BINARY"
encoding="ASCII" name="ENUM_0_TYPE">
         <enumerationValue value="1" name="UN"/>
         <enumerationValue value="2" name="DEUX"/>
         <enumerationValue value="3" name="TROIS"/>
```

```
        <enumerationValue value="4" name="QUATRE"/>
      </enumerationType>
    </typeList>
    <fieldList>
      <field name="VERSION">
        <integerType size="7" computationMode="AT_THE_BEST" encoding="BINARY">
          <integerRange maxValue="100" minValue="0"/>
        </integerType>
        <extension>
          <EXTENSIONS>
            <PARAMETERPROPERTIES dataSource="telemetered"/>
          </EXTENSIONS>
        </extension>
      </field>
      <field name="APID">
        <enumerationType ref="ENUM_0_TYPE"/>
        <extension>
          <EXTENSIONS>
            <PARAMETERPROPERTIES dataSource="telemetered"/>
          </EXTENSIONS>
        </extension>
      </field>
      <field name="PARAM_ENTIER_1">
        <integerType size="16" computationMode="FORCED" encoding="BINARY">
          <integerRange maxConstantNameRef="CST_50" minValue="-30"/>
          <extension>
            <EXTENSIONS>
              <DEFAULTCALIBRATOR calibratorRef="calibrator1"/>
            </EXTENSIONS>
          </extension>
        </integerType>
        <extension>
          <EXTENSIONS>
            <PARAMETERPROPERTIES dataSource="telemetered"/>
          </EXTENSIONS>
        </extension>
      </field>
      <field name="PARAM_ENTIER_2">
        <integerType size="1" computationMode="AT_THE_BEST" encoding="BINARY">
          <integerRange minValue="0"/>
        </integerType>
        <extension>
          <EXTENSIONS>
            <PARAMETERPROPERTIES dataSource="telemetered"/>
            <DEFAULTALARM alarmRef="alarm1"/>
          </EXTENSIONS>
        </extension>
      </field>
      <field name="SIZE_TAB">
        <integerType size="8" computationMode="AT_THE_BEST" encoding="ASCII">
          <integerRange maxValue="100" minValue="0"/>
          <defaultValue>50</defaultValue>
        </integerType>
        <extension>
          <EXTENSIONS>
            <PARAMETERPROPERTIES dataSource="telemetered"/>
```

```
          </EXTENSIONS>
        </extension>
      </field>
      <field name="DEFAULT">
        <characterType>
          <characterRange max="255" min="0"/>
        </characterType>
        <extension>
          <EXTENSIONS>
            <PARAMETERPROPERTIES dataSource="telemetered"/>
          </EXTENSIONS>
        </extension>
      </field>
    </fieldList>
    <root name="SPACE_SYSTEM">
      <recordType/>
      <extension>
        <EXTENSIONS>
          <ROOTNATURE>spaceSystem</ROOTNATURE>
        </EXTENSIONS>
      </extension>
    </root>
    <root name="TELEMETRY">
      <recordType>
        <field name="PACKET_HEADER">
          <recordType>
            <field ref="VERSION"/>
            <field ref="APID"/>
          </recordType>
        </field>
        <field name="PACKET_DATA">
          <recordType>
            <field name="TM1">
              <recordType>
                <field ref="PARAM_ENTIER_1"/>
                <field ref="PARAM_ENTIER_2"/>
              </recordType>
              <optionality discrimined="true">
                <condition function="false">
                  <discriminationAssociation>
                    <discriminedFieldPath>TELEMETRY/PACKET_DATA/TM1</discriminedFieldPath>
                    <conditionValue>TELEMETRY/PACKET_HEADER/APID[UN]</conditionValue>
                  </discriminationAssociation>
                </condition>
              </optionality>
            </field>
            <field name="TM2">
              <recordType>
                <field ref="SIZE_TAB"/>
                <field name="TAB">
                  <arrayType>
                    <repetedField ref="DEFAULT"/>
                    <arrayDimensionRange minSizeValue="1" sizeMode="DISCRETE_REFERENCE">
                      <condition function="false">
                        <discriminationAssociation>
```

```
<discriminedFieldPath>TELEMETRY/PACKET_DATA/TM2/TAB</discriminedFieldPath>
                                        <conditionValue>TELEMETRY/PACKET_DATA/TM2/SIZE_TAB</conditionValue>
                                    </discriminationAssociation>
                                </condition>
                            </arrayDimensionRange>
                        </arrayType>
                    </field>
                </recordType>
                <optionality discrimined="true">
                    <condition function="false">
                        <discriminationAssociation>
                            <discriminedFieldPath>TELEMETRY/PACKET_DATA/TM2</discriminedFieldPath>
                            <conditionValue>TELEMETRY/PACKET_HEADER/APID[DEUX..TROIS]</conditionValue>
                        </discriminationAssociation>
                    </condition>
                </optionality>
            </field>
        </recordType>
    </field>
</recordType>
<extension>
    <EXTENSIONS>
        <ROOTNATURE>telemetry</ROOTNATURE>
    </EXTENSIONS>
</extension>
</root>
<root name="COMMAND">
    <recordType/>
    <extension>
        <EXTENSIONS>
            <ROOTNATURE>command</ROOTNATURE>
        </EXTENSIONS>
    </extension>
</root>
</internalFormat>
```

ANNEXE C : SOURCES DES CLASSES JAVA DE L'EXEMPLE D'ECRITURE

C.1. LA CLASSE « XIFWRITER »

```
package fr.cs.best.modeller.writer;

import java.math.BigInteger;
import java.util.ArrayList;
import java.util.Hashtable;
import java.util.List;

import modeller.common.Debug;
import modeller.common.XifExtensionUtilities;
import modeller.configuration.ModellerConfiguration;
import modeller.core.dataManager.xifManager.XifDataModel;
import modeller.core.dataManager.xifManager.XifDataModelHandler;
import modeller.core.dataManager.xifManager.XifDiscriminantAssociation;
import modeller.core.dataManager.xifManager.XifResource;
import modeller.core.dataManager.xifManager.api.XifArrayType;
import modeller.core.dataManager.xifManager.api.XifCharacterType;
import modeller.core.dataManager.xifManager.api.XifConstant;
import modeller.core.dataManager.xifManager.api.XifEnumeratedType;
import modeller.core.dataManager.xifManager.api.XifEnumerationValue;
import modeller.core.dataManager.xifManager.api.XifField;
import modeller.core.dataManager.xifManager.api.XifIntegerRange;
import modeller.core.dataManager.xifManager.api.XifIntegerType;
import modeller.core.dataManager.xifManager.api.XifNode;
import modeller.core.dataManager.xifManager.api.XifObject;
import modeller.core.dataManager.xifManager.api.XifPhysicalInformation;
import modeller.core.dataManager.xifManager.api.XifRecordType;
import modeller.core.dataManager.xifManager.api.XifStringOrArraySizeType;
import modeller.core.dataManager.xifManager.api.jaxbObjects.CharacterRangeType;
import fr.cs.best.modeller.common.WriterExtensionUtilities;

public class XifWriter {

	private XifResource xifResource;

	private XifDataModel mainDataModel;

	private XifDataModelHandler mainDataModelHandler;

	private static final String XIF_NAME = "new_xif.xif";

	private static final String XIF_DIRECTORY_PATH = "xif/";

	private static final String[] ENUM_VALUES = { "UN", "DEUX", "TROIS", "QUATRE"
};

	/**
	 * Construction et ecriture du fichier XIF a partir de l'exemple
"xif/simple_tm.xif".
```

```
         */
        public void constructAndWriteXIF(){

              try {


        //=======================================================================
                    // INITIALISATION DE LA CONFIGURATION

        //=======================================================================

                    // Chargement de la configuration du Modeller.
                    ModellerConfiguration.loadModellerConfiguration();


        //=======================================================================
                    // CREATION DES OBJETS PRINCIPAUX DE L'API

        //=======================================================================

                    // On créé un nouveau XifResource et on lui associe un nouveau
XifDataModel.
                    xifResource = new XifResource();

                    // Le XifDataModel est créé avec un nom de XIF qui correspond au
nom du fichier XIF et un élément root
                    // est créé par défaut avec ce nom. Le mode "EAST", "M_C" ou encore
"XSD" est passé également en paramètre pour
                    // spécifier le mode du XIF si celui-ci doit être ouvert dans OASIS
Modeller.
                    xifResource.createNewDataModel(XIF_NAME, "M_C");

                    // On récupère le XifDataModel qui est unique dans ce cas et qui
correspond au XIF en mémoire.
                    mainDataModel = xifResource.getXifDataModel();

                    // On récupère le XifDataModelHandler qui fournit un bon nombre
                    // de méthode de création, modification de noeud dans l'arbre XIF.
                    mainDataModelHandler = mainDataModel.getDataModelHandler();


        //=======================================================================
                    // INITIALISATION DES ELEMENTS DE L'ENTETE DU XIF

        //=======================================================================

                    // On lui affecte un répertoire dans lequel sera sauvegardé le XIF.
                    mainDataModel.setDirectory(XIF_DIRECTORY_PATH);

                    // Affectation du targetNameSpace
                    mainDataModel.setTargetNamespace("http://myNameSpace/myXif",
"myxif");

                    // Ajout des informations physiques au noeud general information.
                    setPhysicalInformations("FCSTC000","PC()","16","ROW");


        //=======================================================================
                    // CREATION DES NOEUDS ROOT
```

```
	//=====================================================================

			// On supprime le noeud root créé par défaut.
			mainDataModelHandler.removeRoot(mainDataModel.getRootAt(0));

			// Création et ajout du noeud root TELEMETRY à l'arbre XIF
			//Ajout de l'extension <ROOTNATURE>spaceSystem</ROOTNATURE>
permettant
			// de definir ce noeud root comme etant le noeud "SPACE SYSTEM"
			createRootNode("SPACE_SYSTEM", "spaceSystem");

			// Création et ajout du noeud root TELEMETRY à l'arbre XIF
			//Ajout de l'extension <ROOTNATURE>telemetry</ROOTNATURE>
permettant
			// de definir ce noeud root comme etant le noeud "TELEMETRY"
			XifNode telemetryRootNode = createRootNode("TELEMETRY",
"telemetry");

			// Création et ajout du noeud root TELEMETRY à l'arbre XIF
			//Ajout de l'extension <ROOTNATURE>command</ROOTNATURE> permettant
			// de definir ce noeud root comme etant le noeud "COMMAND"
			createRootNode("COMMAND", "command");


	//=====================================================================
			// CREATION DES EXTENSIONS GLOBALES

	//=====================================================================
			createGlobalExtensions();


	//=====================================================================
			// CREATION DU HEADER

	//=====================================================================
			createHeader(telemetryRootNode);


	//=====================================================================
			// CREATION DE LA DATA

	//=====================================================================
			createData(telemetryRootNode);

			// Sauvegarde du XIF
			xifResource.save();

		} catch (Exception e) {
			Debug.print(e);
		}

	}

	/**
	 * Affecte les informations physiques de la machine dediee aux donnees decrites
par le XIF.
	 */
	private void setPhysicalInformations(String norm, String sendingMachine, String
wordSize, String arrayArranging) {
```

```
            XifPhysicalInformation physicalInfo =
mainDataModel.getPhysicalInformation();
            physicalInfo.setNorm(norm);
            physicalInfo.setSendingMachine(sendingMachine);
            physicalInfo.setWordSize(wordSize);
            physicalInfo.setArrayArranging(arrayArranging);
      }

      /**
       * Cree un noeud root pour l'arbre XIF et lui affecte l'extension specifique
"M_C" permettant
       * de definir la nautre de l'element root.
       * @param rootName le nom de noeud root.
       * @param rootNature la nature du noeud root.
       * @return le noeud root.
       * @throws Exception une exception.
       */
      private XifNode createRootNode(String rootName, String rootNature) throws
Exception {
            XifNode rootNode = mainDataModelHandler.addNewRoot(rootName);
            XifExtensionUtilities.setExtensionValueFor(rootNode.getUserObject(),
rootNature, "ROOTNATURE");
            return rootNode;
      }

      /**
       * Cree le noeud HEADER et sa descendance.
       * @param rootNode le noeud root auquel rattacher le HEADER.
       * @throws Exception une excpetion.
       */
      private void createHeader(XifNode rootNode) throws Exception {

            // Création d'un HEADER contenant une VERSION et un APID
            //=======================================================

            // Pour créer un record dans la liste des type, il faut le nommer.
            // Si l'on souhaite que celui-ci soit affecté en local au XifField,
celui-ci ne doit pas être nommé.
            // Un type nommé appartient à la liste des types internes.
            XifRecordType header =
XifRecordType.createDefaultXifRecordType(mainDataModel);
            XifField fieldHeader = XifField.createDefaultXifField("PACKET_HEADER",
mainDataModel);
            fieldHeader.setType(header);
            mainDataModelHandler.addElement(rootNode, fieldHeader);

            // Creation de l'element VERSION
            //=======================================================

            // Création du type Entier de la version.
            XifIntegerType version =
XifIntegerType.createDefaultXifIntegerType(mainDataModel);
            // Creation de l'intervalle du type entier.
            XifIntegerRange newRange = version.getIntegerRange();
            newRange.setMinValue(BigInteger.ZERO);
            newRange.setMaxValue(BigInteger.valueOf(100));
            // Affectation des attributs du type Entier.
            version.setIntegerRange(newRange);
            version.setBitSizeComputed();
            version.setBinary();
```

```
            version.setSize("7");

            // Création du field de la version
            XifField fieldVersion = XifField.createDefaultXifField("VERSION",
mainDataModel);
            fieldVersion.setType(version);

            // Ajout de l'extension "telemetred" sur le parametre pour specifier la
nature de celui-ci.
            WriterExtensionUtilities.addDataSourceForParameter(fieldVersion,
"telemetered");

            // Ajout du parametre a la liste des parametres.
            mainDataModelHandler.addGlobalField(fieldVersion);

            // Creation du field referencant le parametre global pour le rattacher au
HEADER.
            XifField fieldRefVersion =
XifField.createReferenceForXifField(fieldVersion, mainDataModel);
            XifNode headerNode = rootNode.getChildNodeNamed(fieldHeader.getName());
            mainDataModelHandler.addElement(headerNode, fieldRefVersion);

            // Creation de l'element APID
            //======================================================

            // Création de l'APID et rattachement au HEADER.
            List<XifEnumerationValue> enumValues = new
ArrayList<XifEnumerationValue>();
            for (int i = 0; i < ENUM_VALUES.length; i++) {
                  XifEnumerationValue enumValue =
XifEnumerationValue.createDefaultXifEnumeratedValue(mainDataModel, ENUM_VALUES[i]);
                  Integer value = new Integer(i + 1);
                  enumValue.setValue(value.toString());
                  enumValues.add(enumValue);
            }

            // Un enumere doit toujours etre nomme et donc faire partie de la liste
interne des types du XIF.
            // Il est donc appele par reference dans les field qui utilisent un type
enumere.
            String enumTypeName = "ENUM_0_TYPE";
            // checks if the type is not already processd
            if (enumTypeName != null) {
                  XifObject t =
mainDataModel.getXifResource().getInternalTypeNamed(enumTypeName);
                  if (t != null) {
                        throw new Exception("Creation du type impossible : Le type ["
+ enumTypeName + "] existe deja.");
                  }
            }
            // creates a new type
            XifEnumeratedType apid =
XifEnumeratedType.createDefaultXifEnumeratedType(mainDataModel);

            for (XifEnumerationValue value : enumValues) {
                  try {
                        apid.addEnumeratedValue(value);
                  } catch (Exception e) {
                        throw new Exception(e.getMessage());
                  }
```

```
            }
            apid.setBitSizeComputed();
            apid.setSize(1);
            apid.setBinaryRepresentation();
            apid.setAscii();
            apid.setName(enumTypeName);
            // Ajout du type APID a la liste des tpyes internes.
            mainDataModelHandler.addInternalType(apid);
            XifEnumeratedType apidRefType =
XifEnumeratedType.createReferenceForXifEnumeratedType(apid, mainDataModel);
            // Création du field de l'APID
            XifField filedApid = XifField.createDefaultXifField("APID",
mainDataModel);
            filedApid.setType(apidRefType);
            // Ajout de l'extension "telemetred" sur le parametre pour specifier la
nature de celui-ci.
            WriterExtensionUtilities.addDataSourceForParameter(filedApid,
"telemetered");
            // Ajout du parametre a la liste des parametres.
            mainDataModelHandler.addGlobalField(filedApid);
            // Creation du field referencant le parametre global pour le rattacher au
HEADER.
            XifField fieldRefApid = XifField.createReferenceForXifField(filedApid,
mainDataModel);
            mainDataModelHandler.addElement(headerNode, fieldRefApid);
      }

      /**
       * Cree le noeud DATA et sa descendance.
       * @param rootNode le noeud root auquel rattacher le DATA.
       * @throws Exception une exception.
       */
      private void createData(XifNode rootNode) throws Exception {

            // Creation du record PACKET_DATA et ajout a l'arbre.
            XifRecordType data =
XifRecordType.createDefaultXifRecordType(mainDataModel);
            XifField fieldData = XifField.createDefaultXifField("PACKET_DATA",
mainDataModel);
            fieldData.setType(data);
            mainDataModelHandler.addElement(rootNode, fieldData);


            XifNode dataNode = rootNode.getChildNodeNamed(fieldData.getName());

            // Creation des noeud TM1 et TM2
            createTM1(dataNode);
            createTM2(dataNode);

      }

      /**
       * Cree le noeud TM1 et sa descendance.
       * @param dataNode le noeud auquel rattacher le TM1.
       * @throws Exception une exception.
       */
      private void createTM1(XifNode dataNode) throws Exception {

            // Creation du record TM1 et ajout a l'arbre.
```

```
            XifRecordType tm1 =
XifRecordType.createDefaultXifRecordType(mainDataModel);
            XifField fieldtm1 = XifField.createDefaultXifField("TM1", mainDataModel);
            fieldtm1.setType(tm1);
            mainDataModelHandler.addElement(dataNode, fieldtm1);
            XifNode tm1Node = dataNode.getChildNodeNamed(fieldtm1.getName());

            XifDiscriminantAssociation discAssos = new XifDiscriminantAssociation();
            discAssos.createAssociationTable(fieldtm1,
"TELEMETRY/PACKET_HEADER/APID[UN]", fieldtm1.getDataModel());
            fieldtm1.setConditionAssociation(discAssos);

            // Creation de la constante (max value) "CST_50" du parametre
PARAM_ENTIER_1 et ajout au modele.
            XifConstant cst50 =
mainDataModel.getXifObjectFactory().createConstantType();
            cst50.setName("CST_50");
            cst50.setTypeNameRef(XifConstant.TYPE_NAME_REF_INT);
            cst50.setValue("50");
            mainDataModelHandler.addConstant(cst50);

            // Creation du parametre PARAM_ENTIER_1 et rattachement a la TM1.
            XifIntegerType paramEntier1 =
XifIntegerType.createDefaultXifIntegerType(mainDataModel);
            // Creation de l'intervalle du type entier.
            XifIntegerRange newRange = paramEntier1.getIntegerRange();
            newRange.setMinValue(BigInteger.valueOf(-30));
            newRange.setMaxConstantNameRef("CST_50");
            // Affectation des attributs du type Entier.
            paramEntier1.setIntegerRange(newRange);
            paramEntier1.setBitSizeForced();
            paramEntier1.setBinary();
            paramEntier1.setSize("16");
            // Affectation de la référence vers fonction de calibration portée par le
type.
            WriterExtensionUtilities.addRefToCalibrator(paramEntier1, "calibrator1");
            XifField fieldEntier1 = XifField.createDefaultXifField("PARAM_ENTIER_1",
mainDataModel);
            fieldEntier1.setType(paramEntier1);
            // Ajout de l'extension "telemetred" sur le parametre pour specifier la
nature de celui-ci.
            WriterExtensionUtilities.addDataSourceForParameter(fieldEntier1,
"telemetered");
            // Ajout du parametre a la liste des parametres.
            mainDataModelHandler.addGlobalField(fieldEntier1);
            // Creation du field referencant le parametre global pour le rattacher au
HEADER.
            XifField fieldRefEntier1 =
XifField.createReferenceForXifField(fieldEntier1, mainDataModel);
            mainDataModelHandler.addElement(tm1Node, fieldRefEntier1);

            // Creation du parametre PARAM_ENTIER_2 et rattachement a la TM1.
            XifIntegerType paramEntier2 =
XifIntegerType.createDefaultXifIntegerType(mainDataModel);
            // Creation de l'intervalle du type entier.
            XifIntegerRange newRange2 = paramEntier2.getIntegerRange();
            newRange2.setMinValue(BigInteger.valueOf(-30));
            newRange2.setMaxConstantNameRef("CST_50");
            // Affectation des attributs du type Entier.
            paramEntier2.setIntegerRange(newRange2);
```

```
            paramEntier2.setBitSizeComputed();
            paramEntier2.setBinary();
            paramEntier2.setSize("1");
            XifField fieldEntier2 = XifField.createDefaultXifField("PARAM_ENTIER_2", 
mainDataModel);
            fieldEntier2.setType(paramEntier2);
            WriterExtensionUtilities.addRefToAlarm(fieldEntier2, "alarm1");
            // Ajout de l'extension "telemetred" sur le parametre pour specifier la 
nature de celui-ci.
            WriterExtensionUtilities.addDataSourceForParameter(fieldEntier2, 
"telemetered");
            // Ajout du parametre a la liste des parametres.
            mainDataModelHandler.addGlobalField(fieldEntier2);
            // Creation du field referencant le parametre global pour le rattacher au 
HEADER.
            XifField fieldRefEntier2 = 
XifField.createReferenceForXifField(fieldEntier2, mainDataModel);
            mainDataModelHandler.addElement(tm1Node, fieldRefEntier2);
      }

      /**
       * Cree le noeud TM2 et sa descendance.
       * @param dataNode le noeud auquel rattacher le TM2.
       * @throws Exception une exception.
       */
      private void createTM2(XifNode dataNode) throws Exception {
            // Creation du record TM2 et ajout a l'arbre.
            XifRecordType tm2 = 
XifRecordType.createDefaultXifRecordType(mainDataModel);
            XifField fieldtm2 = XifField.createDefaultXifField("TM2", mainDataModel);
            fieldtm2.setType(tm2);
            mainDataModelHandler.addElement(dataNode, fieldtm2);
            XifNode tm2Node = dataNode.getChildNodeNamed(fieldtm2.getName());

            XifDiscriminantAssociation discAssos = new XifDiscriminantAssociation();
            discAssos.createAssociationTable(fieldtm2, 
"TELEMETRY/PACKET_HEADER/APID[DEUX..TROIS]", fieldtm2.getDataModel());
            fieldtm2.setConditionAssociation(discAssos);

            // Creation du parametre SIZE_TAB et rattachement a la TM2.
            XifIntegerType sizeTab = 
XifIntegerType.createDefaultXifIntegerType(mainDataModel);
            // Creation de l'intervalle du type entier.
            XifIntegerRange newRangeTab = sizeTab.getIntegerRange();
            newRangeTab.setMinValue(BigInteger.ZERO);
            newRangeTab.setMaxValue(BigInteger.valueOf(100));
            // Affectation des attributs du type Entier.
            sizeTab.setIntegerRange(newRangeTab);
            sizeTab.setBitSizeComputed();
            sizeTab.setAscii();
            sizeTab.setSize("8");
            sizeTab.setDefaultValue(BigInteger.valueOf(50));
            XifField fieldSizeTab = XifField.createDefaultXifField("SIZE_TAB", 
mainDataModel);
            fieldSizeTab.setType(sizeTab);
            // Ajout de l'extension "telemetred" sur le parametre pour specifier la 
nature de celui-ci.
            WriterExtensionUtilities.addDataSourceForParameter(fieldSizeTab, 
"telemetered");
            // Ajout du parametre a la liste des parametres.
```

```
            mainDataModelHandler.addGlobalField(fieldSizeTab);
            // Creation du field referencant le parametre global pour le rattacher au
HEADER.
            XifField fieldRefSizeTab =
XifField.createReferenceForXifField(fieldSizeTab, mainDataModel);
            mainDataModelHandler.addElement(tm2Node, fieldRefSizeTab);

            // Creation de l'element repete dans le tableau cree ensuite.
            XifCharacterType defaultType =
XifCharacterType.createDefaultXifCharacterType(mainDataModel);
            // Création de l'intervalle de définition du caractère.
            CharacterRangeType range = defaultType.getCharacterRange();
            range.setMin(0);
            range.setMax(255);

            XifField fieldDefault = XifField.createDefaultXifField("DEFAULT",
mainDataModel);
            fieldDefault.setType(defaultType);
            // Ajout de l'extension "telemetred" sur le parametre pour specifier la
nature de celui-ci.
            WriterExtensionUtilities.addDataSourceForParameter(fieldDefault,
"telemetered");
            // Ajout du parametre a la liste des parametres.
            mainDataModelHandler.addGlobalField(fieldDefault);
            // Creation du field referencant le parametre global pour le rattacher au
TAB.
            XifField fieldRefDefault =
XifField.createReferenceForXifField(fieldDefault, mainDataModel);

            // Creation du tableau TAB et rattachement a la TM2.
            XifArrayType tab = XifArrayType.createDefaultXifArrayType(mainDataModel);
            tab.setRepetedField(fieldRefDefault);

            XifField fieldTab = XifField.createDefaultXifField("TAB", mainDataModel);
            fieldTab.setType(tab);
            mainDataModelHandler.addElement(tm2Node, fieldTab);

            XifDiscriminantAssociation discDim = new XifDiscriminantAssociation();
            discDim.createAssociationTable(fieldTab,
"TELEMETRY/PACKET_DATA/TM2/SIZE_TAB", fieldTab.getDataModel());
            XifStringOrArraySizeType sizeArrayType =
XifStringOrArraySizeType.createDefaultXifStringOrArraySizeType(mainDataModel);
            sizeArrayType.setDynamicSizeMode();
            sizeArrayType.setConditionAssociation(discDim);
            sizeArrayType.setMaxSizeValue(null);
            tab.removeDimensionAt(0);
            tab.addDimension(sizeArrayType);

      }

      /**
       * Creation des extensions globales. On cree ici un calibrator ainsi qu'une
alarme.
       * @throws Exception une exception.
       */
      private void createGlobalExtensions() throws Exception {
            // Creation des valeurs du polynome
            Hashtable<String, String> aList = new Hashtable<String, String>();
            aList.put("0.0", "0");
            aList.put("5.0", "1");
```

```
            // Appel de la methode de creation de la fonction de calibration

      WriterExtensionUtilities.addGlobalPolynomialCalibrator(mainDataModel.getGeneral
Information(), "calibrator1", null, aList);

            // Appel de la methode de creation de la surveillance

      WriterExtensionUtilities.addGlobalGroundAlarm(mainDataModel.getGeneralInformati
on(), "alarm1", null, WriterExtensionUtilities.alarmLevels.warning,
WriterExtensionUtilities.alarmType.numeric, 10, 20, "seconds");
      }

      /**
       * le main de la classe.
       * @param args les parametres passes en ligne de commande.
       */
      public static void main(String[] args) {
            XifWriter writer = new XifWriter();
            writer.constructAndWriteXIF();
      }

}
```

C.2. LA CLASSE « WRITEREXTENSIONUTILITIES »

```
package fr.cs.best.modeller.common;

import java.util.Hashtable;
import java.util.Iterator;
import java.util.Set;

import modeller.common.XifExtensionUtilities;
import modeller.core.dataManager.xifManager.api.XifGeneralInformation;
import modeller.core.dataManager.xifManager.api.XifObject;

import org.w3c.dom.Node;

public class WriterExtensionUtilities {

	public static enum alarmLevels{
		critical, distress, normal, severe, warning, watch;
	}

	public static enum alarmType{
		enumeration, numeric, string;
	}

	/**
	 * Affecte l'extension ""PARAMETERPROPERTIES"" au xifObject si elle n'existe
pas encore
	 * et affecte à cette extension le datasource (type du paramètre).
	 * @param xifObject le type XIF auquel affecter l'extension.
	 * @throws Exception une exception.
	 */
	public static void addDataSourceForParameter(XifObject xifObject, String
dataSourceValue)
	throws Exception {
		try {
			XifExtensionUtilities.setExtensionValueFor(xifObject, null,
"PARAMETERPROPERTIES");
			Node paramProperties =
XifExtensionUtilities.getExtensionFor(xifObject, "PARAMETERPROPERTIES");
			XifExtensionUtilities.createOrSetAttributeNamed(paramProperties,
"dataSource", dataSourceValue);
		} catch (Exception e) {
			throw new Exception(e.getMessage());
		}
	}

	/**
	 * Affecte l'extension "DEFAULTCALIBRATOR" au xifObject si elle n'existe pas
encore
	 * et affecte à cette extension la reference sur le calibrator apparatenant a
la
	 * liste globale des calibrator.
```

```
	 * @param xifObject le type XIF auquel affecter l'extension.
	 * @param calibratorName le nom du calirator a referencer.
	 * @throws Exception une exception.
	 */
	public static void addRefToCalibrator(XifObject xifObject, String 
calibratorName)
	throws Exception {
		try {
			XifExtensionUtilities.setExtensionValueFor(xifObject, null, 
"DEFAULTCALIBRATOR");
			Node defaultCalibrator = 
XifExtensionUtilities.getExtensionFor(xifObject, "DEFAULTCALIBRATOR");
			XifExtensionUtilities.createOrSetAttributeNamed(defaultCalibrator, 
"calibratorRef", calibratorName);
		} catch (Exception e) {
			throw new Exception(e.getMessage());
		}
	}

	/**
	 * Affecte l'extension "DEFAULTALARM" au xifObject si elle n'existe pas encore
	 * et affecte à cette extension la reference sur l'alarme apparatenant a la
	 * liste globale des alarmes.
	 * @param xifObject le type XIF auquel affecter l'extension.
	 * @param alarmName le nom de l'alarme a referencer.
	 * @throws Exception une exception.
	 */
	public static void addRefToAlarm(XifObject xifObject, String alarmName)
	throws Exception {
		try {
			XifExtensionUtilities.setExtensionValueFor(xifObject, null, 
"DEFAULTALARM");
			Node defaultAlarm = 
XifExtensionUtilities.getExtensionFor(xifObject, "DEFAULTALARM");
			XifExtensionUtilities.createOrSetAttributeNamed(defaultAlarm, 
"alarmRef", alarmName);
		} catch (Exception e) {
			throw new Exception(e.getMessage());
		}
	}

	/**
	 * Creer une alarme sol dans la liste des alarmes du noeud 
"generalInformation"
	 * @param aGeneralInformation le noeud "generalInformation".
	 * @param alarmName le nom de l'alarme.
	 * @param alarmDesc la description de l'alarme.
	 * @param alarmLevel le niveau de l'alarme.
	 * @param alarmType le type de l'alarme.
	 * @param minInclusive la valeur minimale inclusive de l'alarme.
	 * @param maxInclusive la valeur maximale inclusive de l'alarme.
	 * @param timeUnits l'unité de temps de l'alarme.
	 * @throws Exception une exception.
	 */
	public static void addGlobalGroundAlarm(XifGeneralInformation 
aGeneralInformation, String alarmName,
```

```
                  String alarmDesc, alarmLevels alarmLevel,
                  alarmType alarmType, int minInclusive,
                  int maxInclusive, String timeUnits) throws Exception {


            Node globAlarmListExt =
XifExtensionUtilities.getNeverNullExtensionFor(aGeneralInformation, "ALARMLIST");

            // creates a new alarm
            Node alarmNode =
XifExtensionUtilities.createAndAddChildNamed(globAlarmListExt, "ALARM");
            // creates name attribute
            XifExtensionUtilities.createOrSetAttributeNamed(alarmNode, "name",
alarmName);
            // creates type attribute
            XifExtensionUtilities.createOrSetAttributeNamed(alarmNode, "type",
alarmType.toString());

            // creates a new definition
            Node definitionNode =
XifExtensionUtilities.createAndAddChildNamed(alarmNode, "DEFINITION");

            switch (alarmType) {
            case numeric : {
                  // creates a new numericAlarm
                  Node numericAlarmNode =
XifExtensionUtilities.createAndAddChildNamed(definitionNode, "NUMERICALARM");

                  // creates a new staticAlarmRanges
                  Node staticAlarmRangesNode =
XifExtensionUtilities.createAndAddChildNamed(numericAlarmNode, "STATICALARMRANGES");

                  // creates a new staticAlarmRange
                  Node staticAlarmRangeNode =
XifExtensionUtilities.createAndAddChildNamed(staticAlarmRangesNode,
"STATICALARMRANGE");
                  // creates alarmLevel attribute

      XifExtensionUtilities.createOrSetAttributeNamed(staticAlarmRangeNode,
"alarmLevel", alarmLevel.toString());

                  // creates a new verifyRange
                  Node verifyRangeNode =
XifExtensionUtilities.createAndAddChildNamed(staticAlarmRangeNode, "VERIFYRANGE");
                  // creates verifyRange attributes
                  XifExtensionUtilities.createOrSetAttributeNamed(verifyRangeNode,
"mininclusive", new Integer(minInclusive).toString());
                  XifExtensionUtilities.createOrSetAttributeNamed(verifyRangeNode,
"maxinclusive", new Integer(maxInclusive).toString());

                  // creates a new timeunits
                  XifExtensionUtilities.createAndAddChildNamed(verifyRangeNode,
"TIMEUNITS", timeUnits);
            }
            case enumeration : {
                  // creates a new enumeration Alarm
                  // ....
```

```
            }
            case string : {
                  // creates a new string Alarm
                  // ....
            }
            }
      }

      /**
       * Creer une fonction de transfert de type calibrateur polynomial dans la
liste
       * globale des calibrateurs du noeud "generalInformation".
       * @param aGeneralInformation le noeud "generalInformation".
       * @param aName le nom du calibrateur.
       * @param aDesc la description du calibrateur.
       * @param aList la liste des couples coefficient/exposant.
       * @throws Exception une exception.
       */
      public static void addGlobalPolynomialCalibrator(XifGeneralInformation
aGeneralInformation, String aName, String aDesc,
                  Hashtable<String, String> aList) throws Exception {
            // récupération de la liste des calibrateurs
            Node globCalExt =
XifExtensionUtilities.getNeverNullExtensionFor(aGeneralInformation,
"CALIBRATORLIST");

            // création du nouveau calibrateur
            Node calNode = XifExtensionUtilities.createAndAddChildNamed(globCalExt,
"CALIBRATOR");

            // ajout du nom du calibrateur
            XifExtensionUtilities.createOrSetAttributeNamed(calNode, "name", aName);

            // ajout de la description
            XifExtensionUtilities.createOrSetAttributeNamed(calNode,
"shortDescription", aDesc);

            // creation du polynome
            Node polynomialNode =
XifExtensionUtilities.createAndAddChildNamed(calNode, "POLYNOMIAL");
            Set<String> keys = aList.keySet();
            for (Iterator<String> iter = keys.iterator(); iter.hasNext();) {
                  String pt = iter.next();
                  // ajout des coefficients et facteurs du polynome
                  Node termNode =
XifExtensionUtilities.createAndAddChildNamed(polynomialNode, "TERM");
                  XifExtensionUtilities.createOrSetAttributeNamed(termNode,
"coefficient", pt);
                  XifExtensionUtilities.createOrSetAttributeNamed(termNode,
"exponent", aList.get(pt));
            }
      }
}
```

ANNEXE D : SOURCE DES CLASSES JAVA DE L'EXEMPLE DE COPIE

```
package fr.cs.best.modeller.copy;

import java.io.File;
import java.util.ArrayList;

import modeller.ModellerServicesInterface;
import modeller.common.Utilities;
import modeller.core.dataManager.xifManager.XifCopiedConditionForDiscriminant;
import modeller.core.dataManager.xifManager.XifCopiedType;
import modeller.core.dataManager.xifManager.XifDataModel;
import modeller.core.dataManager.xifManager.XifDataModelHandler;
import modeller.core.dataManager.xifManager.XifResource;
import modeller.core.dataManager.xifManager.XifTypeManager;
import modeller.core.dataManager.xifManager.api.XifField;
import modeller.core.dataManager.xifManager.api.XifNode;
import modeller.core.dataManager.xifManager.api.XifObject;
import modeller.core.dataManager.xifManager.api.XifRecordType;
import modeller.core.dataManager.xifManager.api.XifUndefinedType;

public class XifCopy {

	private XifResource sourceXifResource;

	private XifDataModel sourceMainDataModel;

	private XifDataModelHandler sourceMainDataModelHandler;

	private XifResource targetXifResource;

	private XifDataModel targetMainDataModel;

	private XifCopiedType tm1Copy;

	private XifNode searchNode = null;

	private static final String FICHIER_XIF = "xif/simple_tm.xif";

	private static final String FICHIER_XIF_COPIE = "xif/simple_tm_copied.xif";

	private static final String FICHIER_XIF_TARGET = "xif/simple_tm_target.xif";

	/**
	 * Traitement de la copie.
	 */
	public void processCopyInXif() {
		try {
			// Chargement du fichier XIF source.
```

```
                  sourceXifResource =
ModellerServicesInterface.loadXmlIf(FICHIER_XIF);
                  sourceMainDataModel = sourceXifResource.getXifDataModel();
                  sourceMainDataModelHandler =
sourceMainDataModel.getDataModelHandler();

                  // recherche du champ père de l'élément à copier.
                  XifNode root = sourceMainDataModel.getNodeFromPath("TELEMETRY");
                  searchPacketData(root, "PACKET_DATA");

                  // Copie de l'élément "TM1", création du champ "TM3" undéfini
                  // puis recopie de l'élément dans le nouveau champ.
                  if (searchNode != null) {
                        XifNode tm1Node = searchNode.getChildNodeNamed("TM1");
                        if (tm1Node != null) {
                              tm1Copy = copyType(searchNode);

                              // Creation d'un element TM3 sur le noeud "PACKET_DATA"
                              XifUndefinedType tm3 =
XifUndefinedType.createDefaultUndefined(sourceMainDataModel);
                              XifField fieldTm3 =
XifField.createDefaultXifField("TM3", sourceMainDataModel);
                              fieldTm3.setType(tm3);
                              sourceMainDataModelHandler.addElement(searchNode,
fieldTm3);
                              XifNode tm3Node = searchNode.getChildNodeNamed("TM3");

                              // Collage de l'élément copié.
                              XifObject typeToPaste = tm1Copy.getCopiedType();
                              sourceMainDataModelHandler.modifyElement(tm3Node,
typeToPaste);
                              if (tm1Copy.getDiscriminantAssociationList() != null) {
                                    ArrayList<XifCopiedConditionForDiscriminant>
listOfDiscriminantAssociationToKeep = tm1Copy.getDiscriminantAssociationList();

      Utilities.completeConditionDiscriminantInTable(tm3Node,
listOfDiscriminantAssociationToKeep);

      Utilities.createConditionDiscriminantOfCopiedType(typeToPaste,
listOfDiscriminantAssociationToKeep, sourceMainDataModel);
                              }
                        }
                  }

                  // Sauvegarde du modèle modifié dans un nouveau XIF.
                  sourceXifResource.saveAs(new File(FICHIER_XIF_COPIE));
            }
            catch (Exception e) {
                  System.out.println("[ERROR] Unable to load the current XIF file.
\n" + e.getMessage());
            }
      }

      /**
       * Effectue une recherche recursive sur le noeud aChild
       * afin de retrouver le champ nommé "PACKET_DATA"
       * @param aChild le noeud à parcourir.
```

```
        * @param fieldName la nom du champ recherché.
        * @throws Exception une exception.
        */
      private void searchPacketData(XifNode aChild, String fieldName) throws
Exception {
            XifObject field = aChild.getUserObject();
            XifObject type = field.getType();
            if (type.isRecordType()) {
                  XifRecordType aRecord = (XifRecordType) type;
                  if (field.getName().equals(fieldName)) {
                        searchNode = aChild;
                  } else {
                        int nbChildren = aRecord.getNumberOfFields();
                        for (int i = 0 ; i < nbChildren ; i++) {
                              XifNode child = aChild.getChildNodeAt(i);
                              searchPacketData(child, fieldName);
                        }
                  }
            }
      }

      /**
       * Copie le type du noeud passé en paramètre et retourne une copie.
       * @param aNode le noeud à coper.
       * @return la copie.
       * @throws Exception une exception.
       */
      protected XifCopiedType copyType(XifNode aNode) throws Exception {
            // Récupération du modèle de données.
            XifDataModel dataModel = aNode.getDataModel();
            // Récupération du champ contenu par le noeud.
            XifObject obj = aNode.getUserObject();
            if (obj.isField()) {
                  // Récupération du type du champ.
                  XifObject type = obj.getType().getResolvedDefinition();
                  // On vérifie que le type n'esy pas un type undéfini
                  // sinon il n'y a pas d'intérêt à effectuer la copie.
                  if (!type.isUndefinedType()) {
                        // Chaque type offre une méthode de copie de lui-même.
                        XifObject copiedType = type.copy(dataModel, true);
                        // On effectue la copie des discriminant si il en existe dans
la structure copiée.
                        ArrayList<XifCopiedConditionForDiscriminant>
discriminantsToKeepList = new ArrayList<XifCopiedConditionForDiscriminant>();
                        Utilities.completeCopiedType(type, copiedType,
discriminantsToKeepList);
                        // On retourne la nouvelle instance de l'élément copié.
                        return new XifCopiedType(copiedType, dataModel,
discriminantsToKeepList, (XifField)obj);
                  }
                  else {
                        throw new Exception("Le type est un type undefini ....");
                  }
            }
            else {
                  throw new Exception("La copie n'est possible que sur un XifField");
            }
```

```
	}


	/**
	 * Traitement de la copie entre deux XIF.
	 */
	public void processCopyBetweenTwoXif() {
		try {

			XifNode copiedNode = null;
			XifNode pastNode = null;

			// Chargement du fichier XIF source.
			sourceXifResource =
ModellerServicesInterface.loadXmlIf(FICHIER_XIF);
			sourceMainDataModel = sourceXifResource.getXifDataModel();

			// Chargement du fichier XIF cible.
			targetXifResource =
ModellerServicesInterface.loadXmlIf(FICHIER_XIF_TARGET);
			targetMainDataModel = targetXifResource.getXifDataModel();


			// recherche du champ père de l'élément à copier.
			XifNode rootSource =
sourceMainDataModel.getNodeFromPath("TELEMETRY");
			searchPacketData(rootSource, "PARAM_ENTIER_1");
			copiedNode = searchNode;

			// Recherche de l'élément cible dans le modele cible.
			XifNode rootTarget =
targetXifResource.getXifDataModel().getNodeFromPath("TELEMETRY");
			searchPacketData(rootTarget, "AN_UNDEFINED");
			pastNode = searchNode;

			// Copie de l'élément source
			XifCopiedType copiedType = copyType(copiedNode);

			// Récupération de l'élément copié.
			XifObject typeToPaste = copiedType.getCopiedType();

			// On vérifie que le type de l'élément copié n'existe pas dans le
XIF cible
			// si celui-ci est nommé.
		    ArrayList<XifObject> typeNamedInCopy = typeToPaste.getTypeNamedIn();
		for (int i = 0; i < typeNamedInCopy.size(); i++) {
			XifObject aType = typeNamedInCopy.get(i);
			if (targetXifResource.getInternalTypeNamed(aType.getName()) != null)
{
				throw new Exception("Le type copié existe déjà dans le modèle cible
....");
			}
		}
		// Le type doit être cloné pour être transporté.
		typeToPaste = XifTypeManager.clone(typeToPaste, targetMainDataModel);
		typeNamedInCopy = typeToPaste.getTypeNamedIn();
		// Ajout du type dans la liste des types du descriptif cible.
```

```
			for (int j = 0; j < typeNamedInCopy.size(); j++) {
	targetMainDataModel.getDataModelHandler().addInternalType(typeNamedInCopy.get(j));
			}

			// Rattachement de l'élément à la cible.
				targetMainDataModel.getDataModelHandler().modifyElement(pastNode, typeToPaste);
				if (copiedType.getDiscriminantAssociationList() != null) {
					ArrayList<XifCopiedConditionForDiscriminant> listOfDiscriminantAssociationToKeep = copiedType.getDiscriminantAssociationList();
					Utilities.completeConditionDiscriminantInTable(pastNode, listOfDiscriminantAssociationToKeep);
	Utilities.createConditionDiscriminantOfCopiedType(typeToPaste, listOfDiscriminantAssociationToKeep, targetMainDataModel);
				}

				// Sauvegarde du modèle modifié dans un nouveau XIF.
				targetXifResource.save();
			}
			catch (Exception e) {
				System.out.println("[ERROR] Unable to load the current XIF file.\n" + e.getMessage());
			}
	}


	/**
	 * Main class for testing
	 * @param args the command line arguments
	 */
	public static void main(String[] args) {
		XifCopy copy = new XifCopy();
		copy.processCopyInXif();
		copy.processCopyBetweenTwoXif();

		System.exit(0);
	}
}
```